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 7 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 1.6 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 an implemntation of the inteface Task. Task extends 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 an implementation of `Task`.
- 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 AbstractTask<String>
```

Here we use the more convenient class AbstractTask, which implements all methods in Task, except for `run()`.
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.setName("Template Job Id");

  // add a task to the job.
  job.add(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.setName(String) is used to give the job a meaningful and readable 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 add(Object task, Object...args). The optional arguments are used when we want to execute other forms of tasks, that are not implementations of Task. 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:

```
try {
  jppfClient = new JPPFClient();
  // ...
} 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<Task<?>> results = jppfClient.submitJob(job);

  // process the results
  processExecutionResults(results);
}
```

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<Task<?>> results = jppfClient.submitJob(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:

```java
public void processExecutionResults(List<JPPFTask> results) {
  // process the results
  for (Task<?> task: results) {
    if (task.getThrowable() != 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:

```java
if (task.getThrowable() != null) {
  System.out.println("An exception was raised: " + task.getThrowable ().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 "startDriver.bat" on Windows or "./startDriver.sh." on Linux/Unix. You should see the following lines printed to the console:

```
driver process id: 2612
management initialized and listening on port 11191
ClientClassServer initialized
NodeClassServer initialized
ClientServer initialized
TasksServer initialized
Acceptor initialized
-   accepting plain connections on port 11111
- accepting secure connections on port 11443
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 "startNode.bat" on Windows or "./startNode.sh." on Linux/Unix. You will then see the following lines printed to the console:

```
node process id: 3336
Attempting connection to the class server at localhost:11111
Reconnected to the class server
JPPF Node management initialized
Attempting connection to the node server at localhost:11111
Reconnected to the node server
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:

```
[client: driver-1] Attempting connection to the class server at localhost:11111
[client: driver-1] Reconnected to the class server
[client: driver-1] Attempting connection to the JPPF task server at localhost:11111
[client: driver-1] Reconnected to the JPPF task server
Execution result: the execution was performed successfully
```

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:

```
Hello, this is the node executing a template JPPF task
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 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:

```java
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) {
    setThrowable(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.setName("Template Job Id");
  // add 10 tasks to the job.
  for (int i=0; i<10; i++) job.add(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:
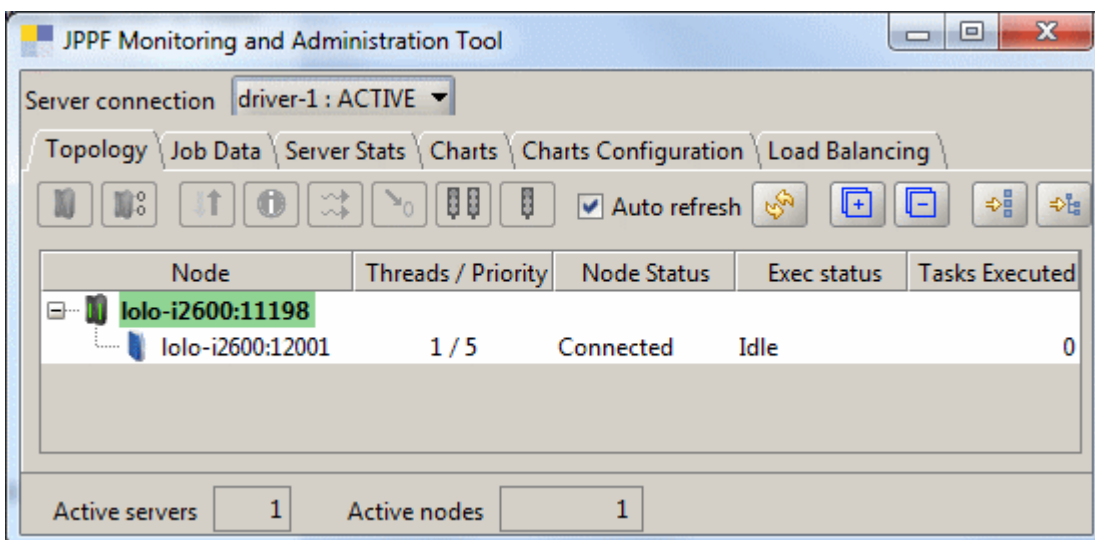
We can see here that a server is started on machine "lolo-quad" and that it has a node attached to it. The color for the server is a health indicator, green meaning that it is running normally and red meaning that it is down.

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 explicitly deploy the application code, since JPPF implicitly 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:
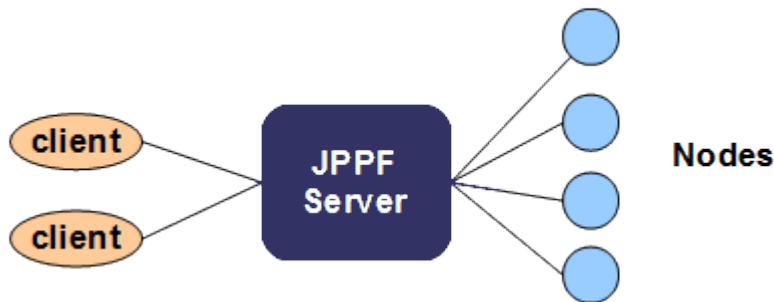
From this picture, we can see that the server plays a central role, and its interactions with the nodes define a *master / worker* architecture, where the server (i.e. master) distributes the work to the nodes (i.e. workers). This also represents the most common topology in a JPPF grid, where each client 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:
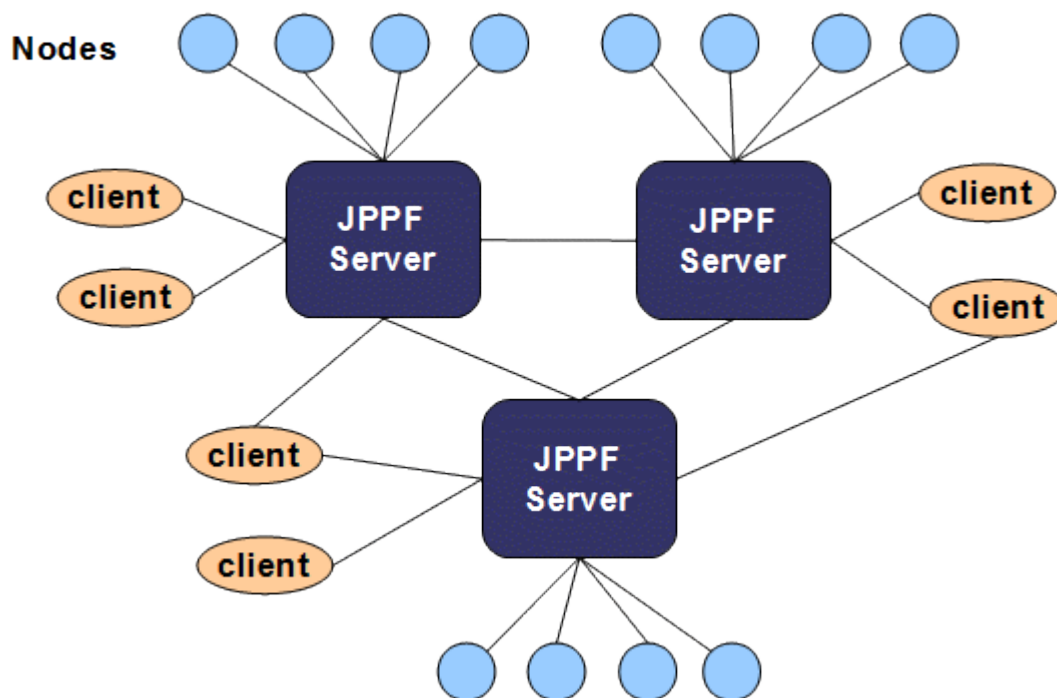
Note how some of the clients are connected to multiple servers, providing failover as well as load balancing capabilities. In addition, and not visible in the previous figure, the nodes have a failover mechanism that will enable them to attach to a different server, should the one they are attached to 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:
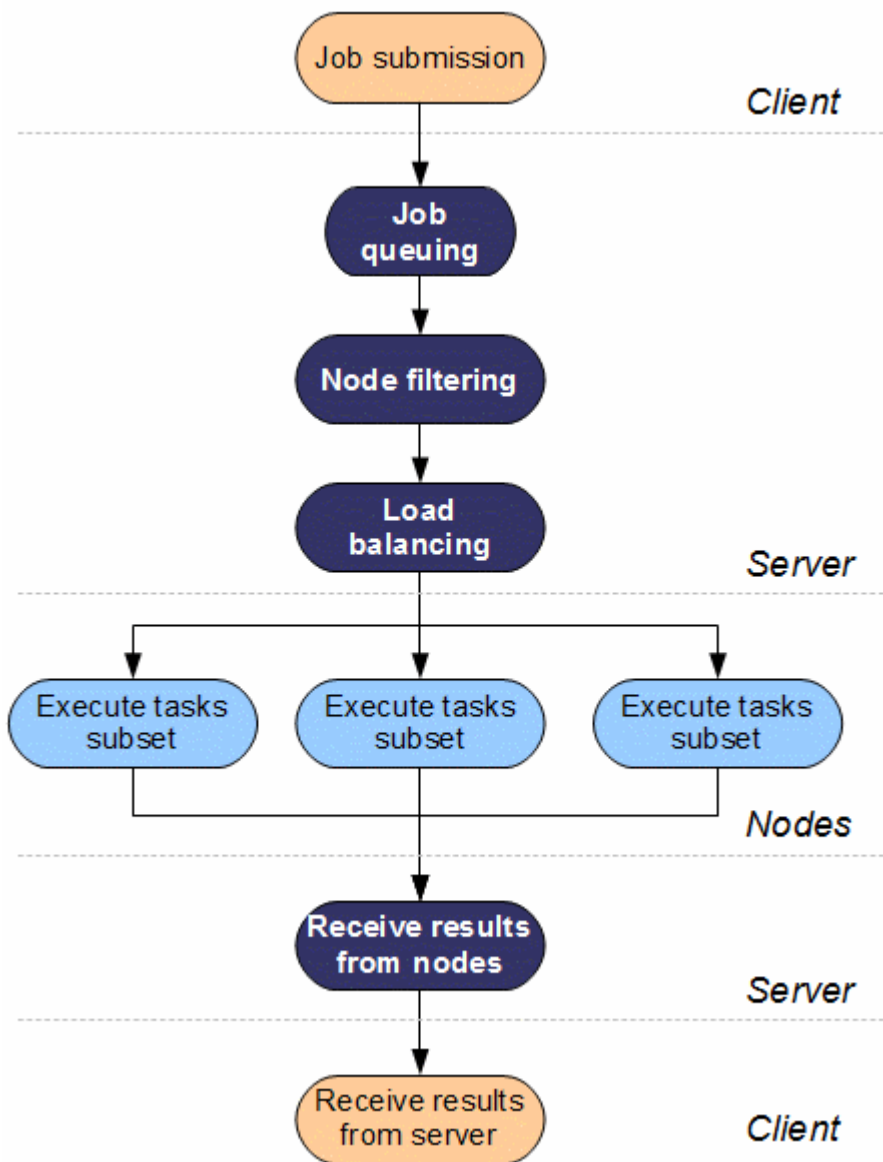
This chart shows the different steps involved in the execution of a job, and where each of them takes place with regards to the grid component boundaries.

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 Task

Task is the base interface 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 Task, are still wrapped by the framework in an implemntation of Task.

JPPFTask is defined as follows:

```
public interface Task<T> extends Runnable, Serializable {
  ...
}
```

We have outlined three important keywords that characterize JPPFTask:

- interface: Task cannot be used directly, it must be implemented/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 serialization mechanism to transform task objects into a form appropriate for networking

The JPPF API provides a convenient abstract implementation of `Task`, which implements all the methods of `Task` except `run()`: to write a real task in your application, you simply extend AbstractTask to implement your own type:

```
public class MyTask extends AbstractTask<Object> {
  @Override
  public void run() {
    // ... your code here ...
  }
}
```

We will now review the functionalities that are inherited from `Task`.

If you are familiar with the JPPF 3.x APIs, please note that the legacy class JPPFTask is now redefined as:

```
public class JPPFTask extends AbstractTask<Object> {
}
```

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 AbstractTask<String> {
  public void run() {
    // ... some code here ...
    setResult("This is the result");
  }
}
```

and later in your application, you would use:

```
String result = 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 explicitly handle exceptions:

- `public void setThrowable(Throwable t)` : store a throwable for later retrieval
- `public Throwable getThrowable()` : retrieve a throwable that was thrown during execution

Here is an example of explicit exception handling:

```
public class MyTask extends AbstractTask<String> {
  public void run() {
    try {
      // ... some code here ...
    } catch(Exception e) {
      setThrowable(e);
    }
  }
}
```

Later on, you can retrieve the throwable as follows:

```
Throwable throwable = myTask.getThrowable();
```

JPPF also automatically 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 `setThrowable()` method. This way, it is always possible for the application to know what happened. The following code shows how JPPF handles uncaught throwables:

```
Task<?> task = ...;
try  {
  task.run();
} catch(Throwable t) {
  task.setThrowable(t);
}
```

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

```
Task<?> task = ...;
if (task.getThrowable() != null) {
  Throwable t = task.getThrowable();
  t.printtStackTrace();
}
```

4.1.1.3 Task life cycle

JPPF provides some options to control a task's life cycle once it has been submitted, including the following:

- task cancellation: this cannot be invoked directly on a task, but is rather invoked as part of cancelling a whole job. If a task is cancelled before its execution starts, then it will never start.
- task timeout: the timeout countdown starts with the task's execution. If a timeout expires before the task starts executing, then the task will not time out.

In all cases, if a task has already completed its execution, it cannot be cancelled 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 is 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 cancelled
`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 illustrating these concepts:

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

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

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

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

A task timeout can be set by using a JPPFSchedule object, which is an immutable object that proposes two constructors:

```
// schedule after a specified duration in milliseconds
public JPPFSchedule(final long duration)
// schedule at a specified fixed date/time
public JPPFSchedule(final String date, final String format)
```

Using a JPPFSchedule, we can thus set and obtain a task timeout using the corresponding accessors:

```
public Task<Object> extends Runnable, Serializable {
  // get the timeout schedule
  public JPPFSchedule getTimeoutSchedule();
  // set a new timeout schedule
  public void setTimeoutSchedule(final JPPFSchedule timeoutSchedule);
}
```

For example:

```
// set the task to expire after 5 seconds
myTask.setTimeout(new JPPFSchedule(5000L));
// set the task to expire on 9/30/2012 at 12:08 pm
myTask.setTimeoutSchedule(new JPPFSchedule("09/30/2012 12:08 PM", "MM/dd/yyyy hh:mm a"));
```

4.1.2 Exception handling - node processing

It is possible that an error occurs while the node is processing a 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.

When such an error occurs, the Throwable that was raised for each task in error is propagated back to the client which submitted the job, and set upon the initial instance of the task in the job. It can then be obtained, upon receiving the execution results, with a call to Task.getThrowable().

4.1.3 Executing code in the client from a task

The Task API provides two methods that will allow a task to send code for execution on the client, and to determine whether the task is executing within a node or within a client with local execution enabled:

```
public interface Task<T> extends Runnable, Serializable {
  // is the task executing in a node or in the client?
  public boolean isInNode()
    // send a callable for execution on the client side
  public <V> V compute(final JPPFCallable<V> callable)
}
```

The method compute() takes a JPPFCallable as input, which is a Serializable extension of the Callable interface and will be executed on the client side.The return value is the result of calling JPPFCallable.call() on the client side.

Example usage:

```
public class MyTask extends AbstractTask<String> {
  @Override
  public void run() {
    String callableResult;
    // if this task is executing in a JPPF node
    if (isInNode()) callableResult = compute(new MyNodeCallable());
    // otherwise if it is executing locally in the JPPF client
    else callableResult = compute(new MyClientallable());
    // set the callable result as this task's result
    setResult(callableResult);
  }

  public static class MyNodeCallable implements JPPFCallable<String> {
    @Override
    public String call() throws Exception {
      return "executed in the NODE";
    }
  }

  public static class MyClientCallable implements JPPFCallable<String> {
    @Override
    public String call() throws Exception {
      return "executed in the CLIENT";
    }
  }
}
```

4.1.4 Sending notifications from a task

Task provides an API which allows tasks to send notifications during their execution:

```
public interface Task<T> extends Runnable, Serializable {
  // Causes the task to send a notification to all listeners
  void fireNotification(Object userObject, boolean sendViaJmx);
}
```

The first parameter *userObject* can be any object provided by the user code. The second parameter *sendViaJmx* specifies whether this notification should also be sent via the node's JPPFNodeTaskMonitorMBean, instead of only to locally registered listeners. If it is `true`, it is recommended that *userObject* be `Serializable`. We will see in further chapters of this documentation how to register local and JMX-based listeners. Let's see here how these listeners can handle the notifications.

Here is an example of JMX listener registered with one or more JPPFNodeTaskMonitorMBean instances:

```
public class MyTaskJmxListener implements NotificationListener {
  @Override
  public synchronized void handleNotification(
    Notification notification, Object handback) {
    // cast to the JPPF notification type
    TaskExecutionNotification notif = (TaskExecutionNotification) notification;
    // get and print the user object
    Object userObject = notif.getUserData();
    System.out.println("received notification with user object = " + userObject);
    // notifications generated by JPPF node always have a  TaskExecutionInfo
    TaskExecutionInfo info = notif.getTaskInformation();
    System.out.println("this notification was sent by "
      + (info == null ? "the user" : "the JPPF node"));
  }
}
```

A local TaskExecutionListener would be like this:

```
public class MyTaskLocalListener extends TaskExecutionListener {
  @Override
  // task completion event sent by the node
  void taskExecuted(TaskExecutionEvent event) {
    TaskExecutionInfo info = event.getTaskInformation();
    System.out.println("received notification with task info = " + info);
  }

  @Override
  // task notification event sent by user code
  void taskNotification(TaskExecutionEvent event) {
    Object userObject = event.getUserObject();
    System.out.println("received notification with user object = " + userObject);
  }
}
```

Consider the following task:

```
public class MyTask extends AbstractTask<String> {
  @Override
  public void run() {
    fireNotification("notification 1", false);
    fireNotification("notification 2", true);
  }
}
```

During the execution of this task, a `MyTaskJmxListener` instance would only receive "notification 2", whereas a `MyTaskLocalListener` instance would receive both "notification 1" and "notification 2".

4.1.5 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:

```java
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:

```java
JPPFJob job = new JPPFJob();
Task<?> task = job.add(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 `Task`, 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.add()` 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 `Task` facilities described in the previous section 4.1.1, except for the callback methods `onCancel()` and `onTimeout()`.

Here is an example using an annotated constructor:

```java
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();
Task<?> task = job.add(MyClass.class, 3, "string arg");
```

Another example using an annotated static method:

```java
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();
Task<?> task = job.add(MyClass.class, 3, "string arg");
```

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

4.1.6 Runnable tasks

Classes that implement <u>java.lang.Runnable</u> 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();
Task<?> 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 `Task`, or annotated with `@JPPFRunnable`, it will be processed as such
- through the task-wrapping mechanism, a Runnable task benefits from the `Task` facilities described in the previous section 4.1.1, except for the callback methods `onCancel()` and `onTimeout()`.

4.1.7 Callable tasks

In the same way as Runnable tasks, classes implementing <u>java.util.concurrent.Callable<V></u> 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();
Task<?> 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 `Task`, 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 `Task` facilities described in the previous section 4.1.1, except for the callback methods `onCancel()` and `onTimeout()`.

4.1.8 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 4.1.5), except for the fact that we need to specify explicitly 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
Task<?> task1 = job.add("MyClass", MyClass.class, 3, "string arg");

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

// add a task using a static method as entry point
Task<?> task3 = job.add("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.add()` 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.add()` 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 `Task` facilities described in the previous section 4.1.1, except for the callback methods `onCancel()` and `onTimeout()`.

4.1.9 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:

```
// launch the process and return its exit code
int launchProcess()
```

This method will block until the process has completed or is destroyed. The process exit code can also be obtained via the following method:

```
// get the process exit code
int getExitCode()
```

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);
    }
  }
}
```

4.1.10 Executing dynamic scripts: ScriptedTask

The class ScriptedTask allows you to execute scripts written in any dynamic language available via the javax.script APIs. It is defined as follows:

```java
public class ScriptedTask<T> extends AbstractTask<T> {
  // Initialize this task with the specified language, script provided as a string
  // and set of variable bindings
  public ScriptedTask(String language, String script, String reusableId,
    Map<String, Object> bindings) throws IllegalArgumentException

  // Initialize this task with the specified language, script provided as a reader
  // and set of variable bindings
  public ScriptedTask(String language, Reader scriptReader, String reusableId,
    Map<String, Object> bindings) throws IllegalArgumentException, IOException

  // Initialize this task with the specified language, script provided as a file
  // and set of variable bindings
  public ScriptedTask(String language, File scriptFile, String reusableId,
    Map<String, Object> bindings) throws IllegalArgumentException, IOException

  // Get the JSR 223 script language to use
  public String getLanguage()

  // Get the script to execute from this task
  public String getScript()

  // Get the unique identifier for the script
  public String getReusableId()

  // Get the user-defined variable bindings
  public Map<String, Object> getBindings()

  // Add the specified variable to the user-defined bindings
  public void addBinding(String name, Object value)

  // Remove the specified variable from the user-defined bindings
  public Object removeBinding(String name)
}
```

Since ScriptedTask is a subclass of AbstractTask, it has all the features that come with it, including life cycle management, error handling, etc. There is a special processing for Throwables raised by the script engine: some engines raise throwables which are not serializable, which may prevent JPPF from capturing them and return them back to the client application. To work around this, JPPF will instantiate a new exception with the same message and stack trace as the original exception. Thus some information may be lost, and you may need to handle these exceptions from within the scripts to retrieve this information.

The reusableId parameter, provided in the constructors, indicates that, if the script engine has that capability, compiled scripts will be stored and reused, to avoid compiling the same scripts repeatedly. In a multithreaded context, as is the case in a JPPF node, multiple compilations may still occur for the script, since it is not possible to guarantee the thread-safety of a script engine, and compiled scripts are always associated with a single script engine instance. Thus, a script may be compiled multiple times, but normally no more than there are processing threads in the node.

Java objects can be passed as variables to the script via the bindings, either in one of the constructors or using the addBinding() and removeBinding() methods. Additionally, a ScriptedTask always adds a reference to itself with the name "jppfTask", or the equivalent in the chosen script language, for instance. $jppfTask in PHP.

The value returned by the script, if any, will be set as the task result, unless it has already been set to a non-null value, by calling jppfTask.setResult(...) from within the script.

For example, in the following Javascript script:

```javascript
function myFunc() {
  jppfTask.setResult('Hello 1');
  return 'Hello 2';
}
myFunc();
```

The result of the evaluation of the script is the string "Hello 2". However, the task result will be "Hello 1", since it was set before the end of the script. If you comment out the first statement of the function (`jppfStat.setResult()` statement), then this time the task result will be the same as the script result "Hello 2".

4.1.11 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 <u>Location</u>, which provides the following methods:

```java
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:

- <u>FileLocation</u> represents a path in the file system
- <u>URLLocation</u> can be used to get data to and from a URL, including HTTP and FTP URLs
- <u>MemoryLocation</u> 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:

```java
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 the job 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
  - an expiration (timeout) date and time
  - an indicator specifying what the server should do when the application is disconnected
- 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
- the ability to receive notifications when the job execution starts and completes
- a persistence manager, to store the job state during execution, and recover its latest saved state on demand, in particular after an application crash

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 Job name and identifier

Each job has a unique identifier (UUID), that allows JPPF to manage and monitor the job while distinguishing it from other jobs. If this identifier is not explicitly specified via a dedicated constructor, JPPF will create one as a string of 32 hexadecimal characters. It is very important that all jobs across an entire JPPF grid have a unique distinct uuid, otherwise there is no guarantee that a job will be executed properly.

Additionally, a job can have a name which doesn't need to be unique, and which is used by the JPPF administration console for display purposes only. You may also use it in your application for logging and tracing. If not set by the user, the name will be by default equal to the uuid.

The class `JPPFJob` provides the following APIs for the job name and uuid:

```
public class JPPFJob
  implements Serializable, JPPFDistributedJob, Iterable<Task<?>> {
  // create a blocking job with the specified uuid
  public JPPFJob(final String jobUuid)

  // get this job's UUID
  public String getUuid()

  // get the user-defined display name for this job
  public String getName()

  // set the user-defined display name for this job
  public void setName(final String name)
}
```

4.2.2 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

```java
// 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, JobSLA jobSLA)
```

Constructors for non-blocking jobs

```java
// 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:

```java
// creates a job with the specified data provider, SLA, blocking indicator
// and execution results listener
public JPPFJob(DataProvider dataProvider, JobSLA 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.

4.2.3 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

```java
public Task<?> add(Object taskObject, Object...args) throws JPPFException
```

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

- an instance of `Task`
- 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 `Task`, 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 `Task`, can be retrieved using the method `Task.getTaskObject()`, as in the following example:

```java
Task<?> task = job.add(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 Task<?> add(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:

```
Task<?> task = job.add("myMethod", new MyPOJO(), 3, "string");
MyPOJO pojo = (MyPOJO) task.getTaskObject();
// we can also set a timeout on the wrapper
task.setTimeoutSchedule(new JPPFSchedule(5000L));
```

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

4.2.4 Inspecting the tasks of a job

JPPFJob provides two ways to get and inspect its tasks: one way is to call the method `getJobTasks()` to obtain the list of tasks, the other is to take advantage of JPPFJob implementing `Iterable<Task<?>>`.

For example, the following two ways to iterate over the tasks in a job are equivalent:

```
JPPFJob myJob = ...;

// get the list of tasks in the job and iterate over them
for (Task<?> task: myJob.getJobTasks()) {
  // do something ...
}

// iterate over the job directly
for (Task<?> task: myJob) {
  // do something ...
}
```

4.2.5 Non-blocking jobs

Jobs can be submitted asynchronously 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 two methods, as illustrated below:

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

  // 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, Task<?>> resultMap = new TreeMap<>();

  // 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) {
      // Insert the tasks in the map, in ascending position order
      for (Task<?> task: event.getTasks()) resultMap.put(task.getPosition(), task);
      // Update the number of pending tasks accordingly
      pendingCount -= event.getTasks().size();
    } else {
      // Reset the state of this listener as if for a new job
      resultMap.clear();
      pendingCount = initialCount;
    }
  }

  // Get the list of results
  public List<Task<?>> getResults() {
    List<Task<?>> results = new ArrayList<>();
    // 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<Task<?>> 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()` methods, so you don't have to write your own synchronization code. Additionally, `JPPFResultCollector` is also always used for blocking jobs.

4.3 Jobs runtime behavior, recovery and failover

4.3.1 Handling of execution results

As we have seen in the previous section, the results of the tasks executions are handed over by an instance of TaskResultListener. Here, it is important to understand that, while JPPF jobs are submitted to the grid as a whole, the results of the tasks execution may be received in multiple chunks, each chunk being made of one or more tasks holding the results.

We can thus say the `TaskResultListener` is an asynchronous receiver, whose job is to:

– handle and store the execution results of the tasks
– ensure the results are in the same order as the tasks initially submitted
– handle errors occurring while receiving the results from the server
– update the state of the job's execution
– optionally handle the persistence of the job's state for later recovery

To store the execution results, the JPPFJob class holds an instance of JobResults, which is accessible via the `getResults()` method. JobResults provides the following API:

```
public class JobResults {
  // Get the current number of received results
  public synchronized int size()

  // Determine whether the job received a result
  // for the task at the specified position
  public synchronized boolean hasResult(final int position)

  // Add the specified results to the job
  public synchronized void putResults(final List<Task<?>> tasks)

  // Get all the tasks received as results for the job
  public synchronized Collection<Task<?>> getAllResults()
}
```

Since JPPFResultCollector holds a reference to the job, it will be able to update the execution results each time it receives a `resultsReceived(TaskResultEvent)` notification.

4.3.2 Failover and job re-submission

When the connection with the JPPF server is broken, the client application becomes unable to receive any more results for the jobs it has submitted and which are still executing. When this happens, the default behavior for the JPPF client is to resubmit the job, so that it will either be sent to another available server, or wait in the client's queue until the connection is re-established.

If the job is using a `TaskResultListener` instance, such as `JPPFResultCollector`, which properly updates the job results via the associated JobResults object, then only the tasks for which no result was received will be sent again to the server. This fault-tolerance mechanism minimizes the impact of losing the connection to the server.

There can be some side effects to this behavior, which should be carefully accounted for when designing your tasks. In effect, the fact that a task result was not received by the client doesn't necessarily mean the task was not executed on a node. This implies that a task may be executed more than once on the grid, as the client has no way of knowing this. In particular, if the task performs persistent operations, such as updating a database or writing to a file system, this may lead to unexpected results whenever the task is executed again.

4.3.3 Job persistence and recovery

The entire state of a job can be persisted by associating a *persistence manager* to the job. A persistence manager is an implementation of the JobPersistence interface, defined as follows:

```
package org.jppf.client.persistence;

public interface JobPersistence<K> {
  // Compute the key for the specified job. All calls to this method
  // with the same job instance should always return the same result.
  K computeKey(JPPFJob job);

  // Get the keys of all jobs in the persistence store
  Collection<K> allKeys() throws JobPersistenceException;

  // Load a job from the persistence store given its key
  JPPFJob loadJob(K key) throws JobPersistenceException;

  // Store the specified tasks of the specified job with the specified key
  // The list of tasks may be used to only store the delta for better performance
  void storeJob(K key, JPPFJob job, List<Task<?>> tasks)
    throws JobPersistenceException;

  // Delete the job with the specified key from the persistence store
  void deleteJob(K key) throws JobPersistenceException;

  // Close this store and release any used resources
  void close();
}
```

As we can see, the persistence manager relies on keys that will allow it to uniquely identify jobs in the persistent store. The type of store is implementation-dependent, and can be any storage device or facility, for example a file system, a database, a cloud storage facility, a distributed cache, etc...

The JPPFJob class provides the following getter and setter for the persistence manager:

```
public class JPPFJob implements Serializable, JPPFDistributedJob {
  // Get the persistence manager
  public <T> JobPersistence<T> getPersistenceManager()

  // Set the persistence manager
  public <T> void setPersistenceManager(final JobPersistence<T> persistenceManager)
}
```

JPPF provides a ready-to-use implementation of JobPersistence: the class <u>DefaultFilePersistenceManager</u>. This implementation stores the jobs on the file system. Each job, with its attributes and tasks, is saved in a single file, using Java serialization. The key associated with each job is the job's uuid (see `JPPFJob.getUuid()` method). It can be instantiated using one of the following constructors:

```java
public class DefaultFilePersistenceManager implements JobPersistence<String> {
  // Initialize with the specified root path, using default file prefix and extension
  public DefaultFilePersistenceManager(File root)

   // Initialize with the specified root path, file prefix and extension
  public DefaultFilePersistenceManager(File root, String prefix, String ext)

  // Initialize with the specified root path, using default file prefix and extension
  public DefaultFilePersistenceManager(String root)

  // Initialize with the specified root path, file prefix and extension
  public DefaultFilePersistenceManager(String root, String prefix, String ext)
}
```

Note, that <u>DefaultFilePersistenceManager</u> will use the serializations scheme configured for the client.
Finally, this persistence manager is shown in action in the <u>Job Recovery</u> related sample.

4.3.4 A full TaskResultListener example

To put together everything we have seen in this chapter, here is a `TaskResultListener` that handles receiving the results, updating the job state, and persisting the job state via a persistence manager. This is, in fact, a simplified version of the code for the class `JPPFResultCollector`, which you can also find in the JPPF SVN repository at this location.

```java
public class MyResultListener implements TaskResultListener {
  protected int count;
  protected int pendingCount = 0;
  protected final JPPFJob job;

  // Initialize this collector with the specified job
  public JPPFResultCollector(JPPFJob job) {
    this.job = job;
    count = job.getJobTasks().size() - job.getResults().size();
    pendingCount = count;
  }

  // Called to notify that the results of a number of tasks
  // have been received from the server
  public synchronized void resultsReceived(TaskResultEvent event) {
    if (event.getThrowable() == null) {
      List<Task<?>> tasks = event.getTasks();
      // update the job's results
      job.getResults().addResults(tasks);
      pendingCount -= tasks.size();
      // notify the threads waiting in waitForResults()
      notifyAll();
      // store the results if a persistence manager is present
      JobPersistence pm = job.getPersistenceManager();
      if (pm != null) {
        try {
          pm.storeJob(pm.computeKey(job), job, tasks);
        } catch (JobPersistenceException e) {
          e.printStackTrace();
        }
      }
    } else {
      // reset this object's state to prepare for job resubmission
      count = job.getJobTasks().size() - job.getResults().size();
      pendingCount = count;
    }
  }

  // Wait until all results of a request have been collected
  public synchronized List<Task<?>> waitForResults() {
    while (pendingCount > 0) {
      try {
        wait();
      } catch(InterruptedException e) {
        e.printStackTrace();
      }
    }
    return getResults();
  }

  // Get the list of final results
  public List<Task<?>> getResults() {
    return new ArrayList<Task<?>>(job.getResults().getAllResults());
  }
}
```

4.3.5 Job start and completion notifications

It is possible to receive notifications for when a job is being started (i.e. sent to the server), when its execution is completed (results have been received for all tasks), when a subset of its tasks is dispatched for execution and when a subset of its tasks has returned from execution. This is done by registering instances of the JobListener interface, which is defined as follows:

```java
// Listener interface for receiving job execution event notifications
public interface JobListener extends EventListener {
  // Called when a job is sent to the server, or its execution starts locally
  void jobStarted(JobEvent event);

  // Called when the execution of a job is complete
  void jobEnded(JobEvent event);

  // Called when a job, or a subset of its tasks, is sent to the server,
  // or to the local executor
  void jobDispatched(JobEvent event);

  // Called when the execution of a subset of a job is complete
  void jobReturned(JobEvent event);
}
```

Please note that jobDispatched() and jobReturned() may be called in parallel by multiple threads, in the case where the JPPF client has multiple connections in its configuration. This happens if the client uses multiple connections to the same server, connections to multiple servers, or a mix of connections to remote servers and a local executor. You will need to synchronize any operations that is not thread-safe within these methods.

In a normal execution cycle, jobStarted() and jobEnded() will be called only once for each job, whereas jobDispatched() and jobReturned() may be called multiple times, depending on the number of available connections, the load-balancing configuration on the client side, and the job's client-side SLA.

Additionally, the built-in job failover mechanism may cause the jobStarted() and jobEnded() callbacks to be invoked multiple times, for instance in the case where the connection to the server is lost, causing the job to be re-submitted.

Note: *it is recommended to only change the job SLA or metadata during the* jobStarted() *notification. Making changes in the other notifications will lead to unpredictable results and may cause the job to fail.*

The notifications are sent as instances of JobEvent, which is defined as:

```java
// Event emitted by a job when its execution starts or completes
public class JobEvent extends EventObject {
  // Get the job source of this event
  public JPPFJob getJob()

  // Get the tasks that were dispatched or returned
  public List<Task<?>> getJobTasks()
}
```

Note that the getTasks() method is only useful for jobDispatched() and jobReturned() notifications. In all other cases, it will return null.

To add or remove listeners, use the related methods in JPPFJob:

```java
public class JPPFJob implements Serializable, JPPFDistributedJob {
  // Add a job listener
  public void addJobListener(JobListener listener)

  // Remove a job listener
  public void removeJobListener(JobListener listener)
}
```

A possible use of these listeners is to "intercept" a job before it is sent to the server, and adjust some of its attributes, such as the SLA specifications, which may vary depending on the time at which the job is started or on an application-dependent context.

4.4 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 <u>DataProvider</u>. Here is the definition of this interface:

```
public interface DataProvider extends Metadata {
  // @deprecated: use getParameter(Object) instead
  <T> T  getValue(final Object key) throws Exception;
  // @deprecated: use setParameter(Object, Object) instead
  void setValue(Object key, Object value) throws Exception;
}
```

As we can see, the two methods in the interface are deprecated, but kept for preserving the compatibility with applications written with a JPPF version prior to 4.0. The actual API is is defined in the <u>Metadata</u> interface as follows:

```
public interface Metadata extends Serializable {
  // Retrieve a parameter in the metadata
  <T> T getParameter(Object key);
  // Return a parameter in the metadata, or a default value if not found
  <T> T getParameter(Object key, T def);
  // Set or replace a parameter in the metadata
  void setParameter(Object key, Object value);
  // Remove a parameter from the metadata
  <T> T removeParameter(Object key);
  // Get the metadata map
  Map<Object, Object> getAll();
  // Clear all the the metadata
  void clear();
}
```

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 Hashtable
DataProvider dataProvider = new MemoryMapDataProvider();
// store the shared object in the data provider
dataProvider.setParameter("myKey", myLargeObject);
JPPFJob = new JPPFJob();
// associate the dataProvider with the job
job.setDataProvider(dataProvider);
job.add(new MyTask());
```

and in a task implementation:

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

4.4.1 MemoryMapDataProvider: map-based provider

`MemoryMapDataProvider` is a very simple implementation of the `DataProvider` interface. It is backed by a `java.util.Hashtable<Object, Object>`. It can be used safely from multiple concurrent threads.

4.4.2 Data provider for non-JPPF tasks

By default, tasks whose class does not implement Task do not have access to the DataProvider that is set on the a job. This includes tasks that implement `Runnable` or `Callable` (including those submitted with a JPPFExecutorService), annotated with @JPPFRunnable, and POJO tasks.

JPPF now provides a mechanism which enables non JPPF tasks to gain access to the `DataProvider`. To this effect, the task must implement the interface DataProviderHolder, defined as follows:

```
package org.jppf.client.taskwrapper;
import org.jppf.task.storage.DataProvider;

// This interface must be implemented by tasks that are not subclasses
// of JPPFTask when they need access to the job's DataProvider
public interface DataProviderHolder {
  // Set the data provider for the task
  void setDataProvider(DataProvider dataProvider);
}
```

Here is an example implementation:

```
public class MyTask
  implements Callable<String>, Serializable, DataProviderHolder {

  // DataProvider set onto this task
  private transient DataProvider dataProvider;

  @Override
  public String call() throws Exception {
    String result = (String) dataProvider.getValue("myKey");
    System.out.println("got value " + result);
    return result;
  }

  @Override
  public void setDataProvider(final DataProvider dataProvider) {
    this.dataProvider = dataProvider;
  }
}
```

Note that the "dataProvider" attribute is set as transient, to prevent the DataProvider from being serialized along with the task when it is sent back to the server after execution. Another way to achieve this would be to set it to `null` at the end of the `call()` method, for instance in a `try {} finally {}` block.

4.5 Job Service Level Agreement

A job service level agreement (SLA) defines the terms and conditions in which a job will be processed. A job carries two distinct SLAs, one which defines a contract between the job and the JPPF server, the other defining a different contract between the job and the JPPF client.

Server and client SLAs have common attributes, which specify:

- the characteristics of the nodes it can run on (server side), or of the channels it can be sent through (client side): the job execution policy
- the time at which a job is scheduled to start
- an expiration date for the job

The attributes specific to the server side SLA are:

- the priority of a job
- whether it is submitted in suspended state
- the maximum number of nodes it can run on
- whether the job is a standard or broadcast job
- whether the server should immediately cancel the job, if the client that submitted it is disconnected

The attributes specific to the client side SLA are:

- the maximum number of channels it can be sent through

A job SLA is represented by the interface `JobSLA` for the server side SLA, and by the interface `JobClientSLA` for the client side SLA. It can be accessed from a job using the related getters and setters:

```java
public class JPPFJob implements Serializable, JPPFDistributedJob {
  // The job's server-side SLA
  public JobSLA getSLA()
  public void setSLA(final JobSLA jobSLA)

  // The job's client-side SLA
  public JobClientSLA getClientSLA()
  public void setClientSLA(final JobClientSLA jobClientSLA)
}
```

Example usage:

```java
JPPFJob myJob = new JPPFJob();
myJob.getClientSLA().setMaxChannels(2);
myJob.getSLA().setPriority(1000);
```

Also note that both interfaces extend the common interface `JobCommonSLA`. We will go into the details of these interfaces in the following sections.

4.5.1 Attributes common to server and client side SLAs

As seen previously, the common attributes for server and client side SLAs are defined by the `JobCommonSLA` interface:

```java
public interface JobCommonSLA extends Serializable {
  // The execution policy
  ExecutionPolicy getExecutionPolicy();
  void setExecutionPolicy(ExecutionPolicy executionPolicy);

  // The job start schedule
  JPPFSchedule getJobSchedule();
  void setJobSchedule(JPPFSchedule jobSchedule);

  // The job expiration schedule
  JPPFSchedule getJobExpirationSchedule();
  void setJobExpirationSchedule(JPPFSchedule jobExpirationSchedule);
}
```

4.5.1.1 Execution policy

An execution policy is an object that determines whether a particular set of JPPF tasks can be executed on a JPPF node (for the server-side SLA) or if it can be sent via a communication channel (for the client-side). It does so by applying the set of rules (or tests) it is made of, against a set of properties associated with the node or channel.

For a fully detailed description of how to create and use execution policies, please read the *Execution policies* section of this development guide.

Example usage:

```java
// define a non-trivial server-side execution policy:
// execute on nodes that have at least 2 threads and whose IPv4 address
// is in the 192.168.1.nnn subnet
ExecutionPolicy serverPolicy = new AtLeast("processing.threads", 2).and(
  new Contains("ipv4.addresses", true, "192.168.1."));
// define a client-side execution policy:
// submit to the client local executor or to drivers whose IPv4 address
// is in the 192.168.1.nnn subnet
ExecutionPolicy clientPolicy = new Equal("jppf.channel.local", true).or(
  new Contains("ipv4.addresses", true, "192.168.1."));
JPPFJob job = new JPPFJob();
// set the server-side policy
job.getSLA().setExecutionPolicy(serverPolicy);
// set the client-side policy
job.getClientSLA().setExecutionPolicy(clientPolicy);
// print an XML representation of the server-side policy
System.out.println("server policy is:\n" + job.getSLA().getExecutionPolicy());
```

4.5.1.2 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:

```java
// 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

```java
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:

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

As a specific date/time

```java
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:

```java
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.getSLA().setJobExpirationSchedule(mySchedule);
```

4.5.2 Server side SLA attributes

A server-side SLA is described by the `JobSLA` interface, defined as:

```java
public interface JobSLA extends JobCommonSLA {
  // Job priority
  int getPriority();
  void setPriority(int priority);

  // Maximum number of nodes the job can run on
  int getMaxNodes();
  void setMaxNodes(int maxNodes);

  // Whether the job is initially suspended
  boolean isSuspended();
  void setSuspended(boolean suspended);

  // Whether the job is a broadcast job
  boolean isBroadcastJob();
  void setBroadcastJob(boolean broadcastJob);

  // Determine whether the job should be canceled by the server
  // if the client gets disconnected
  boolean isCancelUponClientDisconnect();
  void setCancelUponClientDisconnect(boolean cancelUponClientDisconnect);

  // expiration schedule for any subset of the job dispatched to a node
  JPPFSchedule getDispatchExpirationSchedule();
  void setDispatchExpirationSchedule(JPPFSchedule schedule);

  // number of times a dispatched task can expire before it is finally cancelled
  int getMaxDispatchExpirations();
  void setMaxDispatchExpirations(int max);

  // class path associated with the job
  ClassPath getClassPath();
  void setClassPath(ClassPath classpath);
}
```

4.5.2.1 Job priority

The priority of a job determines the order in which the job will be executed by the server. It can be any integer value, such that if jobA.getPriority() > jobB.getPriority() then jobA will be executed before jobB. There are situations where both jobs may be executed at the same time, for instance if there remain any available nodes for jobB after jobA has been dispatched. Two jobs with the same priority will have an equal share (as much as is possible) of the available grid nodes.

The priority attribute is also manageable, which means that it can be dynamically updated, while the job is still executing, using the JPPF administration console or the related management APIs. The default priority is zero.

Example usage:

```java
JPPFJob job1 = new JPPFJob();
job1.getSLA().setPriority(10); // create the job with a non-default priority
JPPFJob job2 = new JPPFJob();
job2.getSLA().setPriority(job1.getSLA().getPriority() + 1); // slightly higher priority
```

4.5.2.2 Maximum number of nodes

The maximum number of nodes attribute determines how many grid nodes a job can run on, at any given time. This is an upper bound limit, and does not guarantee that always this number of nodes will be used, only that no more than this number of nodes will be assigned to the job. This attribute is also non-distinctive, in that it does not specify which nodes the job will run on. The default value of this attribute is equal to Integer.MAX_VALUE, i.e. $2^{31}-1$

The resulting assignment of nodes to the job is influenced by other attributes, especially the job priority and an eventual execution policy. The maximum number of nodes is also a manageable attribute, which means it can be dynamically updated, while the job is still executing, using the JPPF administration console or the related management APIs.

Example usage:

```
JPPFJob job = new JPPFJob();
// this job will execute on a maximum of 10 nodes
job.getSLA().setMaxNodes(10);
```

4.5.2.3 Initial suspended state

A job can be initially suspended. In this case, it will remain in the server's queue until it is explicitly resumed or canceled, or if it expires (if a timeout was set), whichever happens first. A job can be resumed and suspended again any number of times via the JPPF administration console or the related management APIs.

Example usage:

```
JPPFJob job = new JPPFJob();
// this job will be submitted to the server and will remain suspended until
// it is resumed or cancelled via the admin console or management APIs
job.getSLA().setSuspended(true);
```

4.5.2.4 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 interface JobSLA provides the following accessors:

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

To set a job in broadcast mode:

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

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

- a broadcast job is executed on all the nodes connected to the driver, at the time the job is received by the JPPF driver. This includes nodes that are executing another job at that time
- 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

Additionally, if local execution of jobs is enabled for the JPPF client, a broadcast job will not be executed locally. In other words, a broadcast job is only executed on remote nodes.

4.5.2.5 Canceling a job upon client disconnection

By default, if the JPPF client is disconnected from the server while a job is executing, the server will automatically attempt to cancel the job's execution on all nodes it was dispatched to, and remove the job from the server queue. You may disable this behavior on a per-node basis, for example if you want to let the job execute until completion but do not need the execution results. This property is not dynamically manageable.

Example usage:

```
JPPFJob myJob = new JPPFJob();
myJob.getSLA().setCancelUponDisconnect(true);
```

4.5.2.6 Expiration of job dispatches

Definition: *a job dispatch is the whole or part of a job that is dispatched by the server to a node.*

The server-side job SLA enables specifying whether a job dispatch will expire, along with the behavior upon expiration. This is done with a combination of two attributes: a *dispatch expiration schedule*, which specifies when the dispatch will expire, and a *maximum number of expirations* after which the tasks in the dispatch will be cancelled instead of resubmitted. By default, a job dispatch will not expire and the number of expirations is set to zero (tasks are cancelled upon the first expiration, if any).

One possible use for this mechanism is to prevent resource-intensive tasks from bloating slow nodes, without having to cancel the whole job or set timeouts on inidividual tasks.

Example usage:

```
JPPFJob job = new JPPFJob();
// job dispatches will expire if they execute for more than 5 seconds
job.getSLA().setDispatchExpirationSchedule(new JPPFSchedule(5000L));
// dispatched tasks will be resubmitted at most 2 times before they are cancelled.
job.getSLA().setMaxDispatchExpirations(2);
```

4.5.2.7 Setting a class path onto the job

The classpath attribute of the job SLA allows sending library files along with the job and its tasks. Out of the box, this attribute is only used by offline nodes, to work around the fact that offline nodes do no have remote class loading capabilities. The class path attribute, by default empty but not not null, is accessed with the following methods:

```
public interface JobClientSLA extends JobCommonSLA {
  // get / set the class path associated with the job
  ClassPath getClassPath();
  void setClassPath(ClassPath classpath);
}
```

We can see that a class path is represented by the ClassPath interface, defined as follows:

```
public interface ClassPath extends Serializable, Iterable<ClassPathElement> {
  // add an element to this classpath
  ClassPath add(ClassPathElement element);
  ClassPath add(String name, Location<?> location);
  ClassPath add(String name, Location<?> localLocation, Location<?> remoteLocation);

  // remove an element from this classpath
  ClassPath remove(ClassPathElement element);
  ClassPath remove(String name);

  // get an element with the specified name
  ClassPathElement element(String name);

  // get all the elements in this classpath
  Collection<ClassPathElement> allElements();

  // empty this classpath (remove all elements)
  ClassPath clear();

  // is this classpath empty?
  boolean isEmpty();

  // should the node force a reset of the class loader before executing the tasks?
  boolean isForceClassLoaderReset();
  void setForceClassLoaderReset(boolean forceReset);
}
```

Note that one of the add(...) methods uses a ClassPathElement as parameter, while the others use a name with one or two Location objects (see the Location API section). These methods are equivalent. For the last two, JPPF will internally create instances of a default implementation of ClassPathElement (class ClassPathElementImpl). It is preferred to avoid creating ClassPathElement instances, as it makes the code less cumbersome and independent from any specific implementation.

Also note that ClassPath implements Iterable<ClassPathElement>, so that it can be used in for loops:

```
for (ClassPathElement elt: myJob.getSLA().getClassPath()) ...;
```

The ClassPathElement interface is defined as follows:

```
public interface ClassPathElement extends Serializable {
  // get the name of this classpath element
  String getName();
   // get the local (to the client) location of this element
  Location<?> getLocalLocation();
   // get the remote (local to the node) location of this element, if any
  Location<?> getRemoteLocation();
   // perform a validation of this classpath element
  boolean validate();
}
```

JPPF provides a default implementation ClassPathElementImpl which does not perform any validation, that is, its `validate()` method always returns `true`.

Finally, here is an example of how this can all be put together:

```
JPPFJob myJob = new JPPFJob();
ClassPath classpath = myJob.getSLA().getClassPath();
// wrap a jar file into a FileLocation object
Location jarLocation = new FileLocation("libs/MyLib.jar");
// copy the jar file into memory
Location location = jarLocation.copyTo(new MemoryLocation(jarLocation.size()));
// or another way to do this:
location = new MemoryLocation(jarLocation.toByteArray());
// add it as classpath element
classpath.add("myLib", location);
// the following is functionally equivalent:
classpath.add(new ClassPathElementImpl("myLib",  location));
// tell the node to reset the tasks classloader with this new class path
classpath.setForceClassLoaderReset(true);
```

4.5.3 Client side SLA attributes

A client-side SLA is described by the interface `JobClientSLA`, defined as:

```
public interface JobClientSLA extends JobCommonSLA {
  // The maximum number of channels the job can be sent through,
  // including the local executor if any is configured
  int getMaxChannels();
  void setMaxChannels(int maxChannels);
}
```

Note: *since JPPF clients do not have a management interface, none of the client-side SLA attributes are manageable.*

4.5.3.1 Maximum number of execution channels

The maximum number of channels attribute determines how many server connections a job can be sent through, at any given time. This is an upper bound limit, and does not guarantee that this number of channels will always be used. This attribute is also non-specific, since it does not specify which channels will be used. The default value of this attribute is 1.

Using more than one channel for a job enables faster I/O between the client and the server, since the job can be split in multiple chunks and sent to the server via multiple channels in parallel.

Note 1: *when the JPPF client is configured with a single server connection, this attribute has no effect.*
Note 2: *when local execution is enabled in the JPPF client, the local executor counts as one (additional) channel.*
Note 3: *the resulting assignment of channels to the job is influenced by other attributes, especially the execution policy.*

Example usage:

```
JPPFJob job = new JPPFJob();
// use 2 channels to send the job and receive the results
job.getClientSLA().setMaxChannels(2);
```

4.6 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 interface: `JobMetadata`, and can be accessed from the job as follows:

```
JPPFJob job = ...;
JobMetadata metaData = job.getMetadata();
```

`JobMetadata` is defined as follows:

```
public interface JobMetadata extends Metadata {
}
```

As for the data provider, the API is actually defined by the Metadata interface:

```
public interface Metadata extends Serializable {
  // 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 the metadata map
  public Map<Object, Object> getAll()
  // Clear all the the metadata
  void clear();
}
```

Here is an example use:

```
JPPFJob job = ...;
JobMetadata metaData = job.getMetadata();
// 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.

4.7 Execution policies

An execution policy is an object that determines whether a particular set of JPPF tasks can be executed on a JPPF node (for the server-side SLA) or if it can be sent via a communication channel (for the client-side). It does so by applying the set of rules (or tests) it is made of, against a set of properties associated with the node or channel.

There are other uses for execution policies in JPPF, in particular for node selection in some of the management APIs.

The available properties 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
– A special boolean property named "jppf.channel.local" which indicates whether a node (server-side) or communication channel (client-side) is local to the JPPF server or client, respectively.

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' "
– Expressions or scripts written in a script language such as Groovy or JavaScript
– 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 execution policies such as:
"Execute on a node only if the node has at least 256 MB of memory and at least 2 CPUs available"
"Execute the job only in the client's local executor"

In the context of a server-side SLA, an execution policy is sent along with the tasks to the JPPF driver, and evaluated by the driver. It does 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***.

4.7.1 Creating and using an execution policy

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

By API, **using the classes in the <u>org.jppf.node.policy</u> 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);
}
```

4.7.2 Scripted policies

As we have seen earlier, execution policies are objects whose class extends `ExecutionPolicy`. The evaluation of an execution policy is performed by calling its `accepts()` method, which returns either `true` or `false`. A script policy is a special type of policy which can execute a script written in a script language. The result of the evaluation of this script, which must be a boolean, will be the value returned by its accept() method.

By default, JPPF provides engines for Groovy and JavaScript with the Rhino engine, however additional script languages can be added via the service provider interface (SPI).

4.7.2.1 Creating scripted policies

At runtime, a scripted policy is an instance of ScriptedPolicy, which defines the following constructors:

```
public class ScriptedPolicy extends ExecutionPolicy {
  // create with a script read from a string
  public ScriptedPolicy(String language, String script)

  // create with a script read from a reader
  public ScriptedPolicy(String language, Reader scriptReader) throws IOException

  // create with a script read from a file
  public ScriptedPolicy(String language, File scriptFile) throws IOException
}
```

The equivalent XML is as follows:

```
<Script language="_language_">simple script</Script>

<Script language="_language_"><![CDATA[
  a more complex
  script here
]]></Script>
```

As for any other execution policy predicate, scripted policies can be combined with other predicates, using the logical operators AND, OR, XOR and NOT, for instance:

```
// Java
ExecutionPolicy policy = new AtLeast("processing.threads", 2).and(
  new ScriptedPolicy("groovy", "return true"));

<!-- XML equivalent -->
<And>
  <Equal valueType="numeric">
    <Property>processing.threads</Property>
    <Value>2</Value>
  </Equal>
  <Script language="groovy">return true</Script>
</And>
```

The script must either be an expression which resolves to a boolean value, or return a boolean value. For instance, the Groovy statement "return true" and the Groovy expression "true" will work seamlessly.

4.7.2.2 Predefined variable bindings

6 pre-defined variables are made available to the scripts:

- `jppfSystemInfo`: the parameter passed to the policy's accepts() method, its type is JPPFSystemInformation
- `jppfSla`: the server-side SLA of the job being matched, if available, of type JobSLA
- `jppfClientSla`: the client-side SLA of the job being matched, if available, of type JobClientSLA
- `jppfMetadata`: the metadata of the job being matched, if available, of type JobMetadata
- `jppfDispatches`: the number of nodes the job is already disatched to, of type `int`
- `jppfStats`: the server statistics, of type JPPFStatistics

For example, let's look at the following JavaScript script, which determines the node participation based on a jobs priority: if the priority is 1 or less, the job can use no more than 10% of the total number of nodes, if the job priority is 2 then if can use no more than 20%, … up to 90% if the priority is 9 or more:

```
function accepts() {
  // total nodes in the grid from the server statistics
  var totalNodes = jppfStats.getSnapshot("nodes").getLatest();
  // the job priority
  var prio = jppfSla.getPriority();
  // determine max allowed nodes for the job, as % of total nodes
  var maxPct = (prio <= 1) ? 0.1 : (prio >= 9 ? 0.9 : prio / 10.0);
  // return true if current nodes for the job is less than max %
  return jppfDispatches < totalNodes * maxPct;
}

// returns a boolean value
accepts();
```

Let's say this script is stored in a file located at ./policies/NodesFromPriority.js, we could then create an execution policy out of it, with the following code:

```
ScriptedPolicy policy =
  new ScriptedPolicy("javascript", new File("policies/NodesFromPriority.js"));
```

4.7.2.3 Adding available languages

The JPPF scripting APIs rely entirely on the JSR 223 specification, which is implemented in the JDK's javax.script package. This means that JPPF will be able to use any script language made available to the JVM, including the default JavaScript engine (i.e. Rhino in JDK 7 and Nashorn in JDK 8).

Thus to add a new language, all that is needed is to add the proper jar files, which declare a JSR-223 compliant script engine via the documented SPI discovery mechanism. For example, you can add the Groovy language by simply adding groovy-all-x.y.z.jar to the classpath, because it implements the JSR 223 specification (the jar file is located in the JPPF source distribution at **JPPF/lib/Groovy/groovy-all-1.6.5.jar**).

4.7.3 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 <u>CustomPolicy</u>
- 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 mypackage;
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 {
  @Override
  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="mypackage.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 exactly as in this code snippet:

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

Finally, to enable the use of this custom policy, you will need to deploy the corresponding class(es) to both the server's and the client's classpath.

4.8 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.

4.8.1 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 submitted 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`:

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

4.8.2 Resetting the JPPF client

A [JPPFClient](#) can be reset at runtime, to allow the recycling of its server connections, along with dynamic reloading of its configuration. Two methods are provided for this :

```java
public class JPPFClient extends AbstractGenericClient {
  // close this client, reload the configuration, then open it again
  public void reset()

  // close this client, then open it again using the specified confguration
  public void reset(TypedProperties configuration)
}
```

Note that jobs that were already submitted by the client are not lost: they remain queued in the client and will be resubmitted as soon as one or more server connections become available again.

4.8.3 Submitting a job

To submit a job, JPPFClient provides a single method:

```java
public List<Tasks<?>> submitJob(JPPFJob job)
```

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

- blocking job: the `submitJob()` 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: `submitJob()` 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 4.2.5).

4.8.4 Cancelling a job

The ability to cancel a job is provided by JPPFClient's superclass [AbstractGenericClient](#), which provides a `cancelJob()` method, defined as follows:

```java
// superclass of JPPFClient
public abstract class AbstractGenericClient extends AbstractJPPFClient  {
  // cancel the job with the specified UUID
  public boolean cancelJob(final String jobUuid) throws Exception;
}
```

This method will work even if the client is connected to multiple drivers. In this case, it will send the cancel request to all the drivers.

4.8.5 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)
```

4.8.6 Receiving notifications for new and failed 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 4.8.1:

```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:

```
// 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:

```
public class MyClientListener implements ClientListener {
  @Override
  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());
  }

  @Override
  public void connectionFailed(ClientEvent event) {
    JPPFClientConnection connection = event.getConnection();
    System.out.println("Connection " + connection.getName() + " has failed");
  }
}

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

4.8.7 Status notifications for existing 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.

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

```
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 {
  @Override
  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 4.8.6 and modify it to add a status listener to each new connection:

```
final MyStatusListener myListener = new MyStatusListener();
```

```
public MyClientListener implements ClientListener {
  @Override
  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 status events on the new connection
    connection.addClientConnectionStatusListener(new MyStatusListener());
  }

  @Override
  public void connectionFailed(ClientEvent event) {
    JPPFClientConnection connection = event.getConnection();
    System.out.println("Connection " + connection.getName() + " has failed");
    // remove the status listener reference
    connection.removeClientConnectionStatusListener(myListener);
  }
}
```

4.8.8 Switching local execution on or off

The JPPFClient API allows users to dynamically 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()

// Specify 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.

4.9 The ClientWithFailover wrapper class

The class ClientWithFailover enables the setup of connections failover based on the priority of the server connections defined in the client configuration. With this API, the configuration allows you define sophisticated failover strategies, such that a client can always failover to another driver when connection to the current one is broken.

This class provides an API that is very similar to that of JPPFClient:

```java
public class ClientWithFailover implements ClientListener, ClientConnectionStatusListener
{
  // Initialize this client wrapper with the specified array of listeners
  public ClientWithFailover(final ClientListener...listeners)

  // Initialize this client wrapper with the specified uuid and array of listeners
  public ClientWithFailover(final String uuid, final ClientListener...listeners)

  // Submit the specified job
  public List<Task<?>> submitJob(final JPPFJob job) throws Exception

  // Cancel the job with the specified uuid
  public boolean cancelJob(final String uuid) throws Exception

  // Close the underlying client and free its resources
  public void  close()

  // Get the JPPF client to which requests are delegated
  public JPPFClient getClient()
}
```

As you can see, the usage for this class is semantically identical to that of JPPFClient.

Here is an example setup:

In the client configuration we define the following driver connections:

```
jppf.drivers = driver-1 driver-2

driver-1.jppf.server.host = www.host-1.com
driver-1.priority = 10

driver-2.jppf.server.host = www.host-2.com
driver-2.priority = 20
```

Note that we define driver-2 with a higher priority than driver-1: we want the client to submit jobs primarily to driver-2, and failover to driver-1 if the connection to driver-2 fails. For this to happen, we use the following code:

```java
ClientWithFailover client = new ClientWithFailover();
JPPFJob job = …;
List<Task<?>> results = client.submitJob(job);
```

Simply using ClientWithFailover ensures that the failover strategy defined in the configuration is accounted for, and the job is submitted to the active connection with the highest priority. If the driver connection fails while the job is executing, the failover still occurs and the job's unexecuted tasks will be resubmitted to the currently active driver connection.

This feature is not limited to two driver connections: you can have multiple connections with the same priority, as well as a hierarchy of nested connection priorities withotu any size constraint.

4.10 JPPF Executor Services

4.10.1 Basic usage

JPPF 2.2 introduced a new API, that serves as an <u>ExecutorService</u> facade to the JPPF client API. This API consists in a simple class: <u>JPPFExecutorService</u>, 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("Running task id " + id);
  }
}
```

4.10.2 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 throughput 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 timeout 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 sufficiently 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();
```

4.10.3 Configuring jobs and tasks

There is a limitation in the `JPPFExecutorService`, in that if you use only the `ExecutorService` interface which it extends, it does not provide a way to use JPPF-specific features, such as job SLA, metadata or persistence, or task timeout, `onTimeout()` and `onCancel()`.

To overcome this limitation without breaking the semantics of `ExecutorSevice`, `JPPFExecutorService` provides a way to specify the configuration of the jobs and tasks that will be submitted subsequently.

This can be done via the [ExecutorServiceConfiguration](#) interface, which can be accessed from a [JPPFExecutorService](#) instance via the following accessor methods:

```
// Get the configuration for this executor service
public ExecutorServiceConfiguration getConfiguration();

// Reset the configuration for this executor service to a blank state
public ExecutorServiceConfiguration resetConfiguration();
```

[ExecutorServiceConfiguration](#) provides the following API:

```
// Get the configuration to use for the jobs submitted by the executor service
JobConfiguration getJobConfiguration();

// Get the configuration to use for the tasks submitted by the executor service
TaskConfiguration getTaskConfiguration();
```

4.10.3.1 Job configuration

The [JobConfiguration](#) interface is defined as follows:

```
public interface JobConfiguration {
  // Get the service level agreement between the jobs and the server
  JobSLA getSLA();

  // Get the service level agreement between the jobs and the client
  JobClientSLA getClientSLA();

  // Get the user-defined metadata associated with the jobs
  JobMetadata getMetadata();

  // Get/set the persistence manager that enables saving and restoring
  // the state of the jobs
  <T> JobPersistence<T> getPersistenceManager();
  <T> void setPersistenceManager(final JobPersistence<T> persistenceManager);

  // Get/set the job's data provider
  DataProvider getDataProvider();
  void setDataProvider(DataProvider dataProvider);

  // Add or remove a listener to/from the list of job listeners
  void addJobListener(JobListener listener);
  void removeJobListener(JobListener listener);
}
```

As we can see, this provides a way to set the properties normally available to [JPPFJob](#) instances, even though the jobs submiited by a `JPPFExecutorService` are not visible. Any change to the `JobConfiguration` will apply to the next job that will be submitted by the executor and all subsequent jobs.

Here is an example usage:

```
JPPFExecutorService executor = ...;
// get the executor ocnfiguration
ExecutorServiceConfiguration config = executor.getConfiguration();
// get the job configuration
JobConfiguration jobConfig = config.getJobConfiguration();
// set all jobs to expire after 5 seconds
jobConfig.getSLA().setJobExpirationSchedule(new JPPFSchedule(5000L));
```

4.10.3.2 Task configuration

The TaskConfiguration interface can be used to set JPPF-specific properties onto executor service tasks that do not implement Task. It is defined as follows:

```
public interface TaskConfiguration {
  // Get/set the delegate for the onCancel() method
  JPPFTaskCallback getOnCancelCallback();
  void setOnCancelCallback(final JPPFTaskCallback cancelCallback);

  // Get/set the delegate for the onTimeout() method
  JPPFTaskCallback getOnTimeoutCallback();
  void setOnTimeoutCallback(final JPPFTaskCallback timeoutCallback);

  // Get/set the task timeout schedule
  JPPFSchedule getTimeoutSchedule();
  void setTimeoutSchedule(final JPPFSchedule timeoutSchedule);
}
```

This API introduces the concept of a callback delegate, which is used in lieu of the "standard" JPPFTask callback methods, Task.onCancel() and Task.onTimeout(). This is done by providing a subclass of JPPFTaskCallback, which is defined as follows:

```
public abstract class JPPFTaskCallback<T> implements Runnable, Serializable {
  // Get the task this callback is associated with
  public final Task<T> getTask();
}
```

Here is a task configuration usage example:

```
JPPFExecutorService executor = ...;
// get the executor configuration
ExecutorServiceConfiguration config = executor.getConfiguration();
// get the task configuration
TaskConfiguration taskConfig = config.getTaskConfiguration();
// set the task to timeout after 5 seconds
taskConfig.setTimeoutSchedule(new JPPFSchedule(5000L));
// set the onTimeout() callback
taskConfig.setOnTimeoutCallback(new MyTaskCallback());
// A callback that sets a timeout message as the task result
static class MyTaskCallback extends JPPFTaskCallback<String> {
  @Override
  public void run() {
    getTask().setResult("this task has timed out");
  }
}
```

4.10.4 JPPFCompletionService

The JDK package java.util.concurrent provides the interface CompletionService, which represents "*a service that decouples the production of new asynchronous tasks from the consumption of the results of completed tasks*". The JDK also provides a concrete implementation with the class ExecutorCompletionService. Unfortunately, this class does not work with a JPPFExecutorService, as it was not designed with distributed execution in mind.

As a convenience, the JPPF API provides a specific implementation of CompletionService with the class JPPFCompletionService, which respects the contract and semantics defined by the CompletionService interface and which can be used as follows:

```
JPPFExecutorService executor = ...;
JPPFCompletionService<String> completion service =
  new JPPFCompletionService<String>(executor);
MyCallable<String> task = new MyCallable<String>();
Future<String> future = completionService.submit(task);

// ... later on ...
// block until a result is available
future = completionService.take();
String result = future.get();
```

4.11 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`:

```java
public class TypedProperties extends Properties {
  // 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(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 specified as a string. If you wish to programmatically 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:

```java
// get the configuration
TypedProperties props = JPPFConfiguration.getProperties();
// set the connection properties programmatically
props.setProperty("jppf.discovery.enabled", "false");
props.setProperty("jppf.drivers", "driver1");
props.setProperty("driver1.jppf.server.host", "www.myhost.com");
props.setProperty("driver1.jppf.server.port", "11111");

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

4.12 The JPPF statistics API

The statistics in JPPF are handled with objects of type JPPFStatistics, which are a grouping of JPPFSnapshot objects, each snapshot representing a value that is constantly monitored, for instanc the number of nodes connected to a server, or the number of jobs in the server queue, etc.

JPPFSnapshot exposes the following API:

```java
public interface JPPFSnapshot extends Serializable {
  // Get the total cumulated sum of the values
  double getTotal();

  // Get the latest observed value
  double getLatest();

  // Get the smallest observed value
  double getMin();

  // Get the peak observed value
  double getMax();

  // Get the average value
  double getAvg();

  // Get the label for this snapshot
  String getLabel();

  // Get the count of values added to this snapshot
  long getValueCount();
}
```

The label of a snapshot is expected to be unique and enables identifiying it within a JPPFStatistics object.

JPPF implements three different types of snapshots, each with a different semantics for the `getLatest()` method:

– CumulativeSnapshot: in this implementation, `getLatest()` is computed as the cumulated sum of all values added to the snapshot. If values are only added, and not removed, then it will always return the same value as `getTotal()`.
– NonCumulativeSnapshot: here, `getLatest()` is computed as the average of the latest set of values that were added, or the latest value if only one was added.
– SingleValueSnapshot: in this implementation, only `getTotal()` is actually computed, all other methods return 0.

A JPPFStatistics object allows exploring the snapshots it contains, by exposing the following methods:

```java
public class JPPFStatistics implements Serializable, Iterable<JPPFSnapshot> {
  // Get a snapshot specified by its label
  public JPPFSnapshot getSnapshot(String label)

  // Get all the snapshots in this object
  public Collection<JPPFSnapshot> getSnapshots()

  // Get the snapshots in this object using the specified filter
  public Collection<JPPFSnapshot> getSnapshots(Filter filter)

  @Override
  public Iterator<JPPFSnapshot> iterator()

  // A filter interface for snapshots
  public interface Filter {
    // Determines whether the specified snapshot is accepted by this filter
    boolean accept(JPPFSnapshot snapshot)
  }
}
```

Note that, since it implements Iterable<JPPFSnapshot>, a JPPFStatistics object can be used directly in a for loop:

```
JPPFStatistics stats = ...;
for (JPPFSnapshot snapshot: stats) {
  System.out.println("got '" + snapshot.getLabel() + "' snapshot");
}
```

Currently, only the JPPF driver holds and maintains a JPPFStatistics instance. It can be obtained directly with:

```
JPPFStatistics stats = JPPFDriver.getInstance().getStatistics();
```

It can also be obtained remotely via the manegement APIs, as described in the section ***Management and monitoring > Server management > Server-level management and monitoring > Server statistics*** of this documentation.

Additionally, the class JPPFStatisticsHelper holds a set of constants definitions for the labels all all the snapshots currently used in JPPF, along with a number of utility methods to ease the use of statistics:

```
public class JPPFStatisticsHelper {
  // Count of tasks dispatched to nodes
  public static String TASK_DISPATCH = "task.dispatch";

  // ... other constant definitions ...

  // Determine wether the specified snapshot is a single value snapshot
  public static boolean isSingleValue(JPPFSnapshot snapshot)

  // Determine wether the specified snapshot is a cumulative snapshot
  public static boolean isCumulative(JPPFSnapshot snapshot)

  // Determine wether the specified snapshot is a non-cumulative snapshot
  public static boolean isNonCumulative(JPPFSnapshot snapshot)

  // Get the translation of the label of a snapshot in the current locale
  public static String getLocalizedLabel(JPPFSnapshot snapshot)

  // Get the translation of the label of a snapshot in the specified locale
  public static String getLocalizedLabel(JPPFSnapshot snapshot, Locale locale)
}
```

Note: *at this time, only English translations are available.*

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 explicitly 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 either the interface JPPFConfiguration.ConfigurationSource, or the interface JPPFConfiguration.ConfigurationSourceReader, 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()` or getPropertyReader() method, depending on which interface it implements.
   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 Includes in the configuration

A JPPF configuration source, whether as a file or as a plugin, can include other configuration sources by adding one or more "**#!include"** statements in the following format:

```
#!include source_type source_path
```

The possible values for *source_type* are "file", "url" or "class". For each of these values, source_path has a different meaning:

- **when *source_type* is "file"**, *source_path* is the path to a file on the file_system or in the JVM's classpath. If the file exists in both classpath and file system, the file system will have priority over the classpath. Relative paths are interpeted as relative to the JVM's current user directory, as determined by `System.getProperty("user.dir")`.

- **when *source_type* is "url"**, source_path is a URL pointing to a configuration file. It must be a URL such that the JVM can open a stream from it.

- **when *source_type* is "class"**, source_path is the fully qualified class name of a configuration plugin, i.e. an implementation of either JPPFConfiguration.ConfigurationSource or JPPFConfiguration.ConfigurationSourceReader.

Examples:

```
# a config file in the file system
#!include file /home/me/jppf/jppf.properties

# a config file in the classpath
#!include file META-INF/jppf.properties

# a config file obtained from a url
#!include url http://www.myhost.com/jppf/jppf.properties

# a config file obtained from a configuration plugin
#!include class myPackage.MyConfigurationSourceReader
```

Includes can be nested without any limit on the nesting level. Thus, you need to be careful not to introduce cycles in your includes. If that happens, JPPF will catch the resulting StackOverflowException and display an error message in the console output and in the log.

5.3 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:
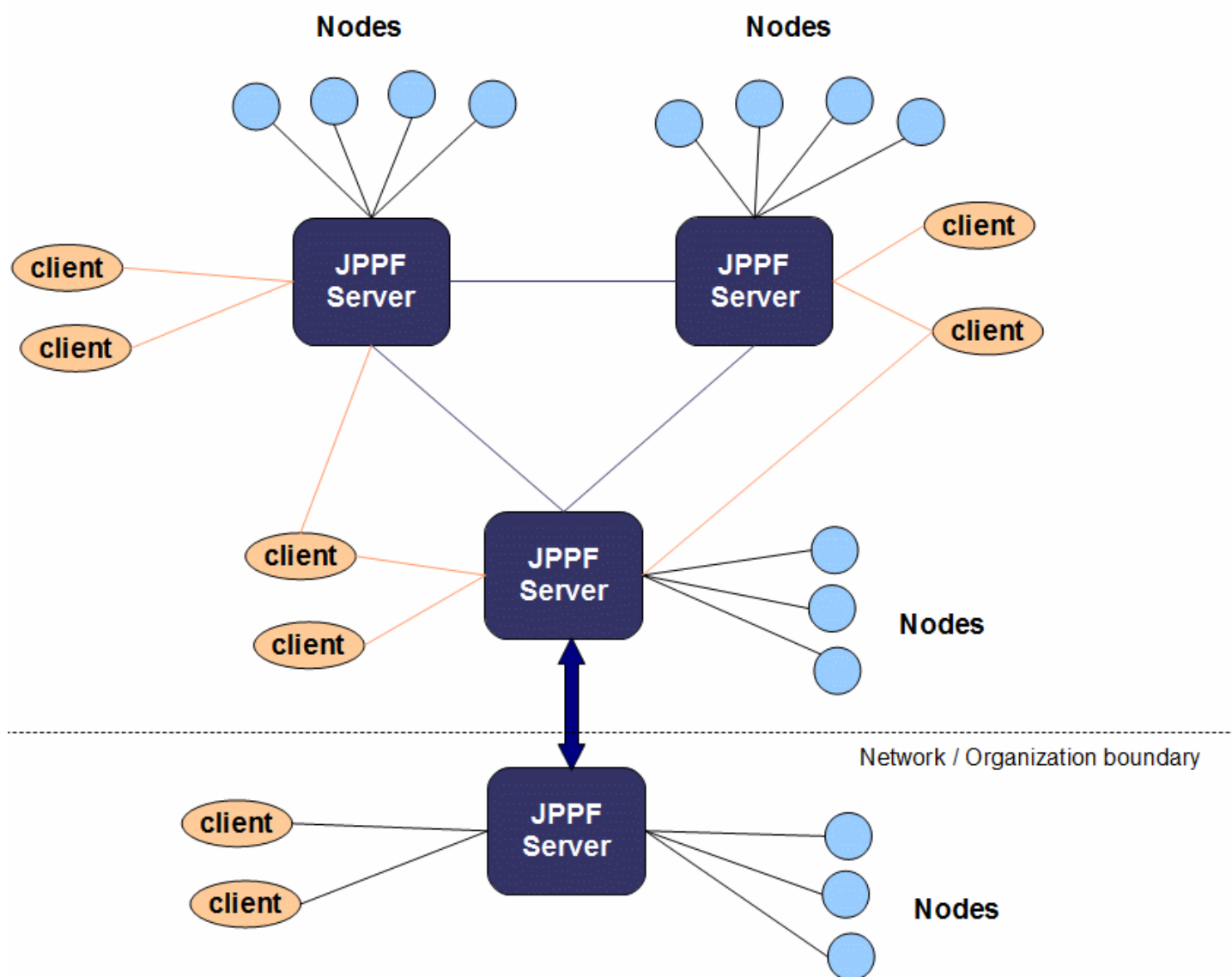
The role of the configuration will be essentially to determine where each component can find the others, and how their interactions will be processed.

5.4 Configuring a JPPF server

5.4.1 Basic network configuration

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

In the configuration file, this property would be defined as follows, with its default value:

```
# JPPF server port
jppf.server.port = 11111
```

Notes:
- not defining this property is equivalent to assigning it its default value.
- dynamic port allocation: when the port number is set to 0, JPPF will dynamically allocate a valid port number. Note that this feature is mostly useful when server discovery is enabled.

5.4.2 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

# IPv4 address inclusion patterns
jppf.discovery.broadcast.include.ipv4 =
# IPv4 address exclusion patterns
jppf.discovery.broadcast.exclude.ipv4 =
# IPv6 address inclusion patterns
jppf.discovery.broadcast.include.ipv6 =
# IPv6 address exclusion patterns
jppf.discovery.broadcast.exclude.ipv6 =
```

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.

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 allows restricting the network interfaces on which to broadcast the server information: the server will only broadcast from the host's addresses that are included and not excluded.

5.4.3 Connecting to other servers

We have seen in section 5.3 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 5.4.2 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
jppf.peer.server_1.server.port = 11111
# connection to server_2
jppf.peer.server_2.server.host = host_2
jppf.peer.server_2.server.port = 11111
```

To connect to each peer, we must define its IP address or host name as well as a port number. Please note that the value we have defined for "`jppf.peer.server_1.server.port`" must be the same as the one defined for "`jppf.server.port`" in server_1's configuration, and the value for "`jppf.peer.server_1.server.port`" must be equal to that of "`jppf.server.port`" in server_1's configuration.

Using manual configuration and server discovery together

It is possible to use the manual configuration simultaneously with the discovery, by adding a special driver name, "jppf_discovery", to the list of manually configured peers:

```
# enable auto-discovery of other peer servers
jppf.peer.discovery.enabled = true
# specifiy both discovery and manually configured drivers
jppf.peerss = jppf_discovery server_1
# connection to server_1
jppf.peer.server_1.server.host = host_1
jppf.peer.server_1.server.port = 11111
```

5.4.4 JMX management configuration

JPPF uses JMX to provide remote management capabilities for the servers, and uses the default JMXMP connector for communication.

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
```

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 explicitly, 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 remote 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.

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.*

5.4.5 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 prefined 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"
- "nodethreads" : each bundle will have at most n * m tasks, where *n* is the number of threads in the node to which the bundle is sent is sent, and *m* is a user-defined parameter

The predefined possible values for the property `jppf.load.balancing.algorithm` are thus: `manual`, `autotuned`, `proportional`, `rl` and `nodethreads`. If not specified, the algorithm defaults to `manual`. For example:

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

In addition to the pre-defined algorithms, it is possible to define your own, as described in section **Extending and Customizing JPPF > Creating a custom load-balancer**.

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.profile = <profile_name>

jppf.load.balancing.profile.<profile_name>.<parameter_1> = <value_1>
...
jppf.load.balancing.profile.<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.profile`. 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.profile = 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.profile = autotuned_profile

# "autotuned" profile
jppf.load.balancing.profile.autotuned_profile.size = 5
jppf.load.balancing.profile.autotuned_profile.minSamplesToAnalyse = 100
jppf.load.balancing.profile.autotuned_profile.minSamplesToCheckConvergence = 50
jppf.load.balancing.profile.autotuned_profile.maxDeviation = 0.2
jppf.load.balancing.profile.autotuned_profile.maxGuessToStable = 50
jppf.load.balancing.profile.autotuned_profile.sizeRatioDeviation = 1.5
jppf.load.balancing.profile.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.profile = proportional_profile

# "proportional" profile
jppf.load.balancing.profile.proportional_profile.initialSize = 5
jppf.load.balancing.profile.proportional_profile.performanceCacheSize = 1000
jppf.load.balancing.profile.proportional_profile.proportionalityFactor = 1
```

"rl" algorithm

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

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

# "rl" profile
jppf.load.balancing.profile.rl_profile.performanceCacheSize = 3000
jppf.load.balancing.profile.rl_profile.performanceVariationThreshold = 0.001
```

"nodethreads" algorithm

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

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

# means that multiplicator * nbThreads tasks will be sent to each node
jppf.load.balancing.profile.nodethreads_profile.multiplicator = 1
```

5.4.6 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 (`-Djppf.config` or `-Djppf.config.plugin`)
- 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
```

5.4.7 Configuring a local node

Each JPPF driver is able to run a single node in its own JVM, called "local node". The main advantage is that the communication between server and node is much faster, since the network overhead is removed. This is particularly useful if you intend to create a pure P2P topology, where all servers communicate with each other and only one node is attached to each server.

To enable a local node in the driver, use the following configuration propoerty, which defaults to "false":

```
jppf.local.node.enabled = true
```

Please note:

- the local node can be configured using the same properties as described in the Node Configuration section, except for the network-related properties, since no network is involved between driver and local node
- for the same reason, the SSL configuration does not apply to a local node

5.4.8 Recovery from hardware failures of 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
```

Important note: *given the implementation of this mechanism, you must be careful to keep some consistency between the server and node settings. As a general rule of thumb, the settings should always satisfy the following constraint:*
serverReaperInterval < nodeMaxRetries * nodeTimeout
This rule applies between any server and node, as well as between any two servers if you enabled communication between servers in your configuration.

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*.

5.4.9 Parallel I/O

The JPPF driver uses several pools of threads to perform network I/O with the nodes, clients and other drivers in parallel. There is a single configuration property that specifies the size of each of these pools:

```
jppf.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()`).

5.5 Node configuration

5.5.1 Server discovery

By default, JPPF nodes are configured to automatically discover active servers on the network. As we have seen in 5.4.2, 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 5.5.2) 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
```

5.5.2 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 a TCP port, as shown in this example:

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

Not defining these properties is equivalent to assigning them their default value (i.e. "localhost" for the host address, 11111 for the port number).

5.5.3 JMX management configuration

JPPF uses JMX to provide remote management capabilities for the nodes, and uses the JMXMP connector for communication.

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
```

These properties have the same meaning and usage as for a server, as described in the driver JMX configuration section.

5.5.4 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
jppf.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
jppf.reconnect.max.time = 60

# time between two connection attempts, in seconds
jppf.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.

5.5.5 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

5.5.6 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.

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
```

Important note: *given the implementation of this mechanism, you must be careful to keep some consistency between the server and node settings. As a general rule of thumb, the settings should always respect the following constraint:*
`serverReaperInterval < nodeMaxRetries * nodeTimeout`
This rule applies between any server and node, as well as between any two servers if you enabled communication between servers.

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*.

5.5.7 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
jppf.processing.threads = 4
```

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

5.5.8 Node process configuration

In the same way as for a server (see 5.4.6 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
```

5.5.9 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.

5.5.10 Class loader resources cache

To avoid uncessary network round trips, the node class loaders store locally the resources found in their extended classpath when one of their methods `getResourceAsStream()`, `getResource()`, `getResources()` or `getMultipleResources()` is called. The type of storage and location of the file-persisted cache can be configured as follows:

```
# type of storage: either "file" (the default) or "memory"
jppf.resource.cache.storage = file
# root location of the file-persisted caches
jppf.resource.cache.dir = some_directory
```

When "file" persistence is configured, the node will fall back to memory persistence if the resource cannot be saved to the file system for any reason. This could happen, for instance, when the file system runs out of space.

For more details, please refer to the "Class Loading In JPPF > Local caching of network resources" section of this documentation.

5.5.11 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.

5.5.12 Offline mode

A node can be configured to run in "offline" mode. In this mode, there will be no class loader connection to the server (and thus no distributed dynamic class loading will occur), and remote management via JMX is disabled. For more details on this mode, please read the documentation section *Deployment and run modes > Offline nodes*.

To turn on the offline mode:

```
# set the offline mode (false by default)
jppf.node.offline = true
```

5.6 Client and administration console configuration

5.6.1 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 5.5.1 for more details.

Additionally, it is possible to specifiy the behavior to adopt, when a driver broadcasts its connection information for multiple network interfaces. In this case, the client may end up creating multiple connections to the same driver, but with different IP addresses. This default behavior can be disabled by setting the following property:

```
# enable or disable multiple network interfaces for each driver
jppf.pool.acceptMultipleInterfaces = false
```

This property is set to `false` by default, meaning that only the first discovered interface for a driver will be taken into account.

5.6.2 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 on which the driver accepts connections
driver-1.jppf.server.port = 11111
```

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.jppf.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.

Note*: the failover based on connection priority will only work if used in conjunction with the* ClientWithFailover *API.See the section* **Development guide > The ClientWithFailover wrapper class** *for details.*

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`.

5.6.3 Using manual configuration and server discovery together

It is possible to use the manual server configuration simultaneously with the server discovery, by adding a special driver name, "jppf_discovery" to the list of manually configured drivers:

```
# enable discovery
jppf.discovery.enabled = true
# specifiy both discovery and manually configured drivers
jppf.drivers = jppf_discovery driver-1
# host for this driver
driver-1.jppf.server.host = my_host
# port for this driver
driver-1.jppf.server.port = 11111
```

5.6.4 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
jppf.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
jppf.reconnect.max.time = 60

# time between two connection attempts, in seconds
jppf.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.

5.6.5 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 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 the client 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.

5.6.6 Local and remote 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. It can also be used for local testing and debugging before going "live".

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.

Finally, the JPPF client also provides the ability to disable remote execution. This can be useful if you want to test the execution of jobs purely locally, even if the server discovery is enabled or the server connection properties would otherwise point to a live JPPF server. To achieve this, simply configure the following:

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

5.6.7 Local execution flow

You can specify how frequently you wish to receive notfiication of locally executed tasks, using either or both of the following parameters:

```
# specifies for how many completed tasks to wait until a notification is sent
jppf.local.execution.accumulation.size = 4

# specifies how long to wait before a notification is sent
# (if any task has completed)
jppf.local.execution.accumulation.time = 100
# Specifies the time unit:
#   n = nanoseconds          M = minutes
#   m = milliseconds         h = hours
#   s = seconds              d = days
jppf.local.execution.accumulation.unit = m
```

If both accumulation time and size are used at the same time, a notification will be sent whenever the size is reached, or the time is reached, whichever happens first. If neither is specified, the tasks wil be returned all at once.

5.6.8 Load-balancing in the client

The JPPF client allows load balancing between local and remote execution. The load balancing configuration is exactly the same as for the driver, which means it uses exactly the same configuration properties, algorithms, parameters, etc... Please refer to the driver load-balancing configuration section for the configuration details. The default configuration, if none is provided, is equivalent to the following:

```
# name of the load balancing algorithm
jppf.load.balancing.algorithm = proportional
# name of the set of parameter values (aka profile) to use for the algorithm
jppf.load.balancing.profile = test
# "proportional" profile
jppf.load.balancing.profile.test.performanceCacheSize = 2000
jppf.load.balancing.profile.test.proportionalityFactor = 1
jppf.load.balancing.profile.test.initialSize = 10
jppf.load.balancing.profile.test.initialMeanTime = 1e9
```

Also note that the load balancing is active even if only remote execution is available. This has an impact on how tasks within a job will be sent tot he server. For instance, if the "manual" algorithm is configured, with a size of 1, this means the tasks in a job will be sent one at a time.

5.7 Configuring SSL/TLS communications

A JPPF grid has the ability to use secure connections between its components. This is done by using the SSL/TLS protocols over network connections, and provides security services such as peer authentication, data encryption and data integrity. This documentation aims at describing how to configure secure connections between JPPF servers, nodes and clients. If you wish to learn details of the SSL/TLS protocols in Java, our recommendation is to read about this in the Java Secure Socket Extension (JSSE) Reference Guide.

Additionally, all downloadable JPPF components now come with a predefined set of SSL configuration files, which can be used *as-is for testing purposes*. These files notably include a truststore and a keystore containing self-signed certificates and private keys. For real-world secure connectivity in JPPF, you will have to provide your own key- and trust- stores, with the proper certificate chains, validated by trusted certificate authorities.

5.7.1 Enabling secure connectivity

5.7.1.1 In the nodes and clients

Nodes and clients use either secure connections, or non-secure connections, but not both at the same time. Thus, this is determined from a single configuration property in their respective configuration file:

```
# Enable SSL. Default value is false (disabled).
# If enabled, only SSL/TLS connections are established
jppf.ssl.enabled = true
```

For a node, this also means that its embedded JMX server will only accept secure connections. Apart from the SSL configuration itself, no other properties are required to enable secure connections: the hosts and ports defined in the configuration or via server discovery will be assumed to be secure. If they are not, no connection will be possible.

5.7.1.2 In the servers

A JPPF server has the ability to accept both secure and non-secure connections, i.e. this is not a single on/off switch as for nodes and clients. Additionally, there are 3 areas of a JPPF server that can be configured separately: "standard" connections from nodes and clients (grid jobs handling and distributed class loader), connections to other servers and embedded JMX server. These are configured via the following properties in the server's configuration file:

```
# Port number to which the server listens for secure connections, defaults to 11443
# A negative value indicates that no secure connection is accepted
jppf.ssl.server.port = 11443

# toggle to enable secure connections to remote peer servers, defaults to false
jppf.peer.ssl.enabled = true

# Enabling JMX features via secure connections, defaults to false
jppf.management.ssl.enabled = true
```

Please note that `jppf.ssl.server.port` (secure port) comes in addition to `jppf.server.port` (non secure) and that both can be used together. For instance, if you wish to only accept secure connections, you will have to disable the non-secure connection by specifying a negative port number:

```
# disable non-secure connections
jppf.server.port = -1
# enable secure connections
jppf.ssl.server.port = 11443
```

As for the non-secure port, assigning a value of 0 will cause JPPF to dynamically allocate a valid port number.

In a similar way you can use either JMX secure or non-secure connections, or both:

```
# Enabling JMX via non-secure connections, defaults to true
jppf.management.enabled = false
# Enabling JMX via secure connections, defaults to false
jppf.management.ssl.enabled = true
# Secure JMX server port
jppf.management.ssl.port = 11193
```

5.7.2 Locating the SSL configuration

The SSL configuration is loaded separately from the JPPF configuration itself. The effect of this is that it is harder to find for a remote application, and it will not appear in the JPPF monitoring tools and APIs, the goal being to avoid providing information about how JPPF is secured, which would defeat the purpose of securing it in the first place.

5.7.2.1 Configuration as a file or classpath resource

To specify the location of the SSL configuration as a file, you can use the `jppf.ssl.configuration.file` property in the JPPF configuration file of the driver, node or client:

```
# location of the SSL configuration in the file system or classpath
jppf.ssl.configuration.file = config/ssl/ssl.properties
```

The lookup for the specified file or resource is performed first in the file system, then in the classpath. This allows you for instance to embed a configuration file in a jar file, with the possibility to override it with another file.

Relative paths are relative to the current working directory as specified by `System.getProperty("user.dir")`.

5.7.2.2 Configuration as an external source

JPPF provides a more sophisticated way to locate its SSL configuration, which requires the implementation of a specific plugin. This is useful in situations where a configuration file is not considered secure enough, or if you need to load the configuration from a centralized location, for instance if you run JPPF in a cloud environment and want to fetch the configuration via a cloud storage facility such as Amazon's S3.

This is done via the `jppf.ssl.configuration.source` property:

```
# SSL configuration as an arbitrary source. Value is the fully qualified name
# of an implementation of java.util.concurrent.Callable<InputStream> with optional
# space-separated arguments
jppf.ssl.configuration.source = implementation_of_Callable<InputStream> arg1 ... argN
```

where *implementation_of_Callable<InputStream>* is the fully qualified name of a class which implements the interface Callable<InputStream> and which must have either a noarg constructor, or a (String...args) vararg constructor.

For example, the predefined JPPF plugin FileStoreSource is implemented as follows:

```
package org.jppf.ssl;

import java.io.InputStream;
import java.util.concurrent.Callable;
import org.jppf.utils.FileUtils;

// A secure store source that uses a file as source
public class FileStoreSource implements Callable<InputStream> {

  // Optional arguments that may be specified in the configuration
  private final String[] args;

  public FileStoreSource(final String... args) throws Exception {
    this.args = args;
    if ((args == null) || (args.length == 0))
      throw new SSLConfigurationException("missing parameter: file path");
  }

  @Override
  public InputStream call() throws Exception {
    // lookup in the file system, then in the classpath
    InputStream is = FileUtils.getFileInputStream(args[0]);
    if (is == null)
      throw new SSLConfigurationException("could not find file " + args[0]);
    return is;
  }
}
```

We can then use it in the configuration:

```
jppf.ssl.configuration.source = org.jppf.ssl.FileStoreSource config/ssl/ssl.properties
```

which is in fact equivalent to:

```
jppf.ssl.configuration.file = config/ssl/ssl.properties
```

5.7.3 SSL configuration properties

These properties are defined in the SSL configuration file and represent the information required to create and initialize SSLContext, SSLSocket and SSLEngine objects.

5.7.3.1 SSLContext protocol

This is the protocol name used in SSLContext.getInstance(String protocol). It is defined as:

```
# SSLContext protocol, defaults to SSL
jppf.ssl.context.protocol = SSL
```

A list of valid protocol names is available here.

5.7.3.2 Enabled protocols

This is the list of supported protocol versions, such as returned by SSLEngine.getEnabledProtocols(). It is defined as a list of space-separated names:

```
# list of space-separated enabled protocols
jppf.ssl.protocols = SSLv2Hello SSLv3
```

A list of valid protocol versions is available here.

5.7.3.3 Enabled cipher suites

This is the list of supported protocol versions, such as returned by SSLEngine.getEnabledCipherSuites(). It is defined as a list of space-separated names:

```
# enabled cipher suites as a list of space-separated values
jppf.ssl.cipher.suites = SSL_RSA_WITH_RC4_128_MD5 SSL_RSA_WITH_RC4_128_SHA
```

A list of supported cipher suites is available here.

5.7.3.4 Client authentication

The client authentication mode is determined by calling the methods SSLEngine.getWantClientAuth() and SSLEngine.getNeedClientAuth(). It is defined as:

```
# client authentication mode
# possible values: none | want | need
jppf.ssl.client.auth = none
```

5.7.3.5 Key store and associated password

As for the location of the SSL configuration, there are two ways to specify the location of a keystore:

```
# path to the key store on the file system
jppf.ssl.keystore.file = config/ssl/keystore.ks
# an implementation of Callable<InputStream> with optional space-separated arguments
jppf.ssl.keytstore.source = org.jppf.ssl.FileStoreSource config/ssl/keystore.ks
```

Note that, if both properties are defined, JPPF will first attempt to load the key store from the defined source, then from the specified file path.

In a similar fashion, there are two ways to specify the key store's password: either as a clear text password, or as a password source. This can be done as follows:

```
# keystore password in clear text
jppf.ssl.keystore.password = password
# keystore password from an arbitrary source
# the source is an implementation of Callable<char[]> with optional parameters
jppf.ssl.keystore.password.source = org.jppf.ssl.PlainTextPassword password
```

5.7.3.6 Trust store and associated password

The trust store and its password are defined in the same way as for the key store:

```
# path to the trust store on the file system
jppf.ssl.truststore.file = config/ssl/truststore.ks
# an implementation of Callable<InputStream> with optional space-separated arguments
jppf.ssl.truststore.source = org.jppf.ssl.FileStoreSource config/ssl/truststore.ks

# keystore password in clear text
jppf.ssl.truststore.password = password
# keystore password from an arbitrary source
# the source is an implementation of Callable<char[]> with opptional parameters
jppf.ssl.truststore.password.source = org.jppf.ssl.PlainTextPassword password
```

5.7.3.7 Special case: distinct driver trust stores for nodes and clients

In the case when the JPPF driver has mutual authentication enabled (jppf.ssl.client.auth = need"), it might be desirable to use distinct trust stores for the certificates of the nodes and the clients that will connect to the driver. This can be done as follows:

```
# specify that a separate trust store must be used for client certificates
jppf.ssl.client.distinct.truststore = true

# path to the client trust store on the file system
jppf.ssl.client.truststore.file = config/ssl/truststore_cli.ks
# an implementation of Callable<InputStream> with optional space-separated arguments
jppf.ssl.client.truststore.source = o.j.s.FileStoreSource config/ssl/truststore_cli.ks

# keystore password in clear text
jppf.ssl.client.truststore.password = password
# keystore password from an arbitrary source
# the source is an implementation of Callable<char[]> with opptional parameters
jppf.ssl.client.truststore.password.source = org.jppf.ssl.PlainTextPassword password
```

In this configuration, a client will not be able to authenticate with the server, even if it uses the same keystore and password as the nodes.

5.7.4 Full SSL configuration example

```
# SSLContext protocol, defaults to SSL
jppf.ssl.context.protocol = SSL
# list of space-separated enabled protocols
jppf.ssl.protocols = SSLv2Hello SSLv3
# enabled cipher suites as a list of space-separated values
jppf.ssl.cipher.suites = SSL_RSA_WITH_RC4_128_MD5 SSL_RSA_WITH_RC4_128_SHA
# client authentication mode; possible values: none | want | need
jppf.ssl.client.auth = none

# path to the key store on the file system.
jppf.ssl.keystore.file = config/ssl/keystore.ks
# keystore password in clear text
jppf.ssl.keystore.password = password

# the trust store location as an arbitrary source:
# an implementation of Callable<InputStream> with optional space-separated arguments
jppf.ssl.truststore.source = org.jppf.ssl.FileStoreSource config/ssl/truststore.ks
# truststore password as an arbitrary source:
# an implementation of Callable<char[]> with optional space-separated arguments
jppf.ssl.truststore.password.source = org.jppf.ssl.PlainTextPassword password
```

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, each node provides 2 MBeans that can be accessed remotely using a JMXMP remote connector with the JMX URL "`service:jmx:jmxmp://host:port`", 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 JPPFNodeAdminMBean.MBEAN_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:

```
public interface JPPFNodeAdminMBean extends JPPFAdminMBean {
  // Get the latest state information from the node.
  public JPPFNodeState state() throws Exception;
}
```

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

```
public class JPPFNodeState implements Serializable {
  // 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()
}
```

6.1.1.2 Updating the execution thread pool properties

```
public interface JPPFNodeAdminMBean extends JPPFAdminMBean {
  // Set the size of the node's execution thread pool.
  public void updateThreadPoolSize(Integer size) throws Exception;

  // Update the priority of all execution threads.
  public void updateThreadsPriority(Integer newPriority) throws Exception;
}
```

6.1.1.3 Shutting down and restarting the node

```
public interface JPPFNodeAdminMBean extends JPPFAdminMBean {
  // Restart the node.
  public void restart() throws Exception;

  // Shutdown the node.
  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

```
public interface JPPFNodeAdminMBean extends JPPFAdminMBean {
  // Reset the node's executed tasks counter to zero.
  public void resetTaskCounter() throws Exception;

  // Reset the node's executed tasks counter to the specified value.
  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

```
public interface JPPFAdminMBean extends Serializable {
  // Get detailed information about the node's JVM properties, environment variables,
  // memory usage, available processors andavailable storage space.
  JPPFSystemInformation systemInformation() throws Exception;
}
```

This method returns an object of type <u>JPPFSystemInformation</u>, 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:

```
public class JPPFSystemInformation implements Serializable {
  // 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

```
public interface JPPFAdminMBean extends Serializable {
  // Cancel the job with the specified uuid. The requeue parameters determines
  // whether the job should be requeued on the server side or not.
  public void cancelJob(String jobUuid, 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 Updating the node's configuration properties

```
public interface JPPFAdminMBean extends Serializable {
  // Update the configuration properties of the node. The reconnect parameter
  // specifies whether the node should disconnect then reconnect to the driver
  // after updating the properties.
  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.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:

```
public interface JPPFNodeTaskMonitorMBean extends NotificationEmitter {
  // 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 uuid 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()
}
```

TaskExecutionNotification also inherits the method `getUserData()`, which returns the object specified by the user code when calling Task.fireNotification(Object, boolean) with the second parameter set to true.

6.1.3 Node maintenance

MBean name : "**org.jppf:name=node.maintenance,type=node**".
This is also the value of the constant `JPPFNodeMaintenanceMBean.MBEAN_NAME`

This MBean provides operations for the maintenance of a node. It exposes the interface `JPPFNodeMaintenanceMBean` defined as follows:

```
public interface JPPFNodeMaintenanceMBean extends Serializable {
  // object name for this MBean
  String MBEAN_NAME = "org.jppf:name=node.maintenance,type=node";

  // request a reset of the resource caches of all the JPPF class loaders
  // maintained by the node
  void requestResourceCacheReset() throws Exception;
}
```

Please note that the method `requestResourceCacheReset()` does not perform the reset immediately. Instead, it sets an internal flag, and the reset will take place when it is safe to do so, as part of the node's life cycle. The outcome of the reset operation is that the temporary files created by the JPPF class loaders wil be deleted, freeing space in the temporary files folder.

6.1.4 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.

6.1.4.1 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, no network connection involved) MBean server, use the no-arg constructor:

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

To connect to a **remote** MBean server, use the constructor specifying 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 ...;
```

6.1.4.2 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();
```

6.1.4.3 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(
  JPPFNodeAdminMBean.MBEAN_NAME, "state", (Object[]) null, (String[]) null);
int nbTasks = state.getNbTasksExecuted();
// get the total CPU time used
long cpuTime = (Long) wrapper.invoke(JPPFNodeTaskMonitorMBean.MBEAN_NAME,
  "getTotalTaskCpuTime", (Object[]) null, (String[]) null);
```

6.1.4.4 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.MBEAN_NAME, JPPFNodeTaskMonitorMBean.class);

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

6.1.4.5 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.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.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();
```

6.1.5 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:

```java
// 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, 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, each server provides 2 MBeans that can be accessed remotely using a JMXMP remote connector with the JMX URL "`service:jmx:jmxmp://host:port`", 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 <u>JPPFDriverAdminMBean.MBEAN_NAME</u>.

This MBean's role is to perform management and monitoring of the server. It exposes the <u>JPPFDriverAdminMBean</u> 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 provides statistics on execution performance, network overhead, server queue behavior, number of connected nodes and clients:

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
  // Get the latest statistics snapshot from the JPPF driver
  public JPPFStatistics statistics() throws Exception;
}
```

This method returns an object of type <u>JPPFStatistics</u>. We invite you to read the dedicated section in ***Development Guide > The JPPF statistics API*** for the full details of its usage.

Additionally, you can reset the server statistics using the following method:

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
  // Reset the JPPF driver statistics
  public void resetStatistics() throws Exception;
}
```

6.2.1.2 Stopping and restarting the server

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
  // Perform a shutdown or restart of the server. The server stops after
  // the specified shutdown delay, and restarts after the specified restart delay
  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. The server will only restart after the restart delay if it is at least equal to zero, otherwise it simply shuts down and cannot be restarted remotely anymore.

6.2.1.3 Managing the nodes attached to the server

The driver MBean allows monitoring and managing the nodes attached tot he driver with the following two methods:

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
  // Request the JMX connection information for all nodes attached to the server.
  public Collection<JPPFManagementInfo> nodesInformation() throws Exception;

  // Get the number of nodes currently attached to the server.
  public Integer nbNodes() throws Exception;
}
```

The <u>JPPFManagementInfo</u> 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> {
  // Get the host on which the driver or node is running
  public synchronized String getHost();

  // Get the port on which the node's JMX server is listening
  public synchronized int getPort()

  // Get the driver or node's unique id
  public String getUuid()

  // Determine whether this is a driver connected as a node to another driver
  public boolean isPeer()

  // Determine whether this information represents a real node
  public boolean isNode()

  // Determine whether this is a driver
  public boolean isDriver()

  // Determine whether communication is be secured via SSL/TLS
  public boolean isSecure()

  // Determine whether this information represents a local node
  public boolean isLocal()

  // Determine whether the node is active or inactive
  public boolean isActive()
}
```

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();
}
```

Additionally, if all you need is the number of nodes attached to the server, then simply calling the `nbNodes()` method will be much more efficient in terms of CPU usage and network traffic.

6.2.1.4 Monitoring idle nodes

The JPPF driver MBean provides two methods to gather information on idle nodes:

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
  // Request the JMX connection information for the
  // currently idle nodes attached to the server.
  public Collection<JPPFManagementInfo> idleNodesInformation() throws Exception;

  // Get the number of currently idle nodes attached to the server.
  public Integer nbIdleNodes() throws Exception;
}
```

`idlesNodesInformation()` is similar to `nodesInformation()` except that it provides information only for the nodes that are currently idle. If all you need is the number of idle nodes, then it is much less costly to call `nbIdleNodes()` instead.

6.2.1.5 Switching the nodes active state

From the server's perspective, the nodes can be considered either active or inactive. When a node is "active" the server

will take it into account when scheduling the execution of the jobs. Inversely, when it is "inactive", no job can be scheduled to execute on this node. In this case, the node behaves as if it were not part of the JPPF grid for job scheduling purposes, while still being alive and manageable.

This provides a way to disable nodes without the cost of terminating the corresponding remote process. For example, this provides a lot of flexibility in how the workload can be balanced among the nodes: sometimes you may need to have more nodes with less processing threads each, while at other times you could dynamically setup less nodes with more processing threads.

This can be done with the following method:

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
  // Toggle the activate state of the specified nodes
  public void toggleActiveState(NodeSelector selector) throws Exception;
}
```

This method acts as an on/off switch: nodes in the 'active' state will be deactivated, whereas nodes in the 'inactive' state will be activated. Also note that this method uses a node selector to specifiy which nodes it applies to.

6.2.1.6 Load-balancing settings

The driver management MBean provides 2 methods to dynamically obtain or change the server's load balancing settings:

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
  // Obtain the current load-balancing settings.
  public LoadBalancingInformation loadBalancerInformation() throws Exception;
}
```

This method returns an object of type LoadBalancingInformation, defined as follows:

```
public class LoadBalancingInformation implements Serializable {
  // Get the name of the algorithm
  public String getAlgorithm()
  // Get the algorithm's parameters
  public TypedProperties getParameters()
  // Get the names of all available algorithms
  public List<String> getAlgorithmNames()
}
```

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:

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
  // Change the load-balancing 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 the profile name "jppf".

6.2.1.7 Testing an execution policy

This feature allows you to compute the number of nodes that match a specific execution policy. This enables testing whether a job, holding this execution policy as part of its SLA, would be executed, before submitting it:

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
  // Compute the number of nodes matching the specified policy.
  public Integer matchingNodes(ExecutionPolicy policy) throws Exception;
}
```

6.2.2 Job-level management and monitoring

MBean name: "**org.jppf:name=jobManagement,type=driver**"
This is also the value of the constant DriverJobManagementMBean.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 jobUuid) throws Exception;
  // Suspend the job with the specified id
  public void suspendJob(String jobUuid, Boolean requeue) throws Exception;
  // Resume the job with the specified id
  public void resumeJob(String jobUuid) throws Exception;
  // Update the maximum number of nodes a job can run on
  public void updateMaxNodes(String jobUuid, Integer maxNodes) throws Exception;
  // Update the priority of a job
  void updatePriority(String jobUuid, Integer newPriority);
  // 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 jobUuid) 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 jobUuid) throws Exception;
  // Update the priority of a job
  void updatePriority(String jobUuid, Integer newPriority);
}
```

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 dispatched 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:

```
public interface DriverJobManagementMBean extends NotificationEmitter {
  // Cancel the job with the specified id.
  public void cancelJob(String jobUuid) 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.

```
public interface DriverJobManagementMBean extends NotificationEmitter {
  // Suspend the job with the specified uuid
  public void suspendJob(String jobUuid, 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.

```
public interface DriverJobManagementMBean extends NotificationEmitter {
  // Resume the job with the specified uuid
  public void resumeJob(String jobUuid) 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

```
public interface DriverJobManagementMBean extends NotificationEmitter {
  // Update the maximum number of nodes a job can run on.
  public void updateMaxNodes(String jobUuid, 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 Updating the priority of a job

```
public interface DriverJobManagementMBean extends NotificationEmitter {
  // Update the priority of a job
  void updatePriority(String jobUuid, Integer newPriority);
}
```

This method dynamically updates the priority of job specified via its uuid. The update takes effect immediately

6.2.2.4 Job introspection

The management features allow users to query and inspect the jobs currently queued or executing in the server. This can be done using two related methods of the jobs management MBean:

```
public interface DriverJobManagementMBean extends NotificationEmitter {
  // Get the set of uuids for all the jobs currently queued or executing
  public String[] getAllJobIds() throws Exception;
  // Get an object describing the job with the specified uuid
  public JobInformation getJobInformation(String jobUuid) throws Exception;
  // Get a list of objects describing the sub-jobs of a job,
  // and the nodes to which they were dispatched
  public NodeJobInformation[] getNodeInformation(String jobUuid) throws Exception;
}
```

The `getAllJobIds()` method returns the UUIDs of all the jobs currently handled by the server. These UUIDs can be directly used with the other methods of the job management MBean.

The method `getJobInformation()` retrieves information about the state of a job in the server. This method returns an object of type <u>JobInformation</u>, defined as follows:

```
public class JobInformation implements Serializable {
  // the job's name
  public String getJobName()
  // 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()
}
```

The method `getNodeInformation()` also allows to obtain information about all the sub-jobs of a job that are disptached to remote nodes. The return value is an array of objects of type <u>NodeJobInformation</u>, 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 <u>JobInformation</u> and <u>JPPFManagementInfo</u>, 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.

6.2.2.5 Job notifications

Whenever a job-related event occurs, the job management MBean will emit a notification of type <u>JobNotification</u>, 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 <u>JobEventType</u> 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 complexity of JMX programming. This API is implemented by the class JMXDriverConnectionWrapper, which 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.

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. 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 jobUuids = 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(
  JPPFDriverAdminMBean.MBEAN_NAME, "statistics", (Object[]) null, (String[]) null);
int nbNodes = stats.getNodes().getLatest();
```

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

Here is an example notification listener implementing the <u>NotificationListener</u> 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, Object handback) {
    String message = notification.getMessage();
    String toDisplay = handback.toString() + ": " + message;
    System.out.println(toDisplay);
  }
}
```

6.3 Nodes management and monitoring via the driver

JPPF provides support for forwarding JMX requests to the nodes, along with receiving notifications from them, via the JPPF driver's JMX server. Which nodes are impacted is determined by a user-provided node selector.

This brings two major benefits:

- this allows managing and monitoring the nodes in situations where the nodes are not reachable from the client, for instance when the client and nodes are on different networks or subnets
- the requests and notifications forwarding mechanism automatically adapts to node connection and disconnection events, which means that if new nodes are started in the grid, they will be automatically enrolled in the forwarding mechanism, provided they match the node selector

6.3.1 Node selectors

All the node forwarding APIs make use of *node selectors* to conveniently specify which nodes they apply to. A node selector is an instance of the NodeSelector interface, defined as follows:

```java
// Marker interface for selecting nodes
public interface NodeSelector extends Serializable {
  // Constant for a selector which accepts all nodes
  NodeSelector ALL_NODES = new AllNodesSelector();

  // Selects all nodes
  public final static class AllNodesSelector implements NodeSelector {
    // Default constructor
    public AllNodesSelector()
  }

  // Selects nodes based on an execution policy
  public final static class ExecutionPolicySelector implements NodeSelector {
    // Initialize this selector with an execution policy
    public ExecutionPolicySelector(final ExecutionPolicy policy)
    // Get the execution policy to use to select the nodes
    public ExecutionPolicy getPolicy()
  }

  // Selects nodes based on their uuids
  public final static class UuidSelector implements NodeSelector {
    // Initialize this selector with a collection of node UUIDs
    public UuidSelector(final Collection<String> uuids)
    // Initialize this selector with an array of node UUIDs
    public UuidSelector(final String...uuids)
    // Get the collection of uuids of the nodes to select
    public Collection<String> getUuidList()
  }
}
```

We can see that three types of selector are available:

– NodeSelector.AllNodesSelector will select all the nodes currently attached to the server. Rather than creating instances of this class, you also use the predefined constant NodeSelector.ALL_NODES.
– NodeSelector.ExecutionPolicySelector uses an execution policy, such as can be set upon a JPPF job's SLA to perform the selection of the nodes
– NodeSelector.UuidSelector will only select nodes whose UUID is part of the collection or array of specified UUIDs

Note that the node selection dynamically adjusts to the JPPF grid topology, or in other words two distinct selections with the same selector instance may return a different set of nodes: when new nodes are added to the grid or existing nodes are terminated, these changes in the toplogy will be automatically taken into account by the selection mechanism.

6.3.2 Forwarding management requests

The request forwarding mechanism is based on a built-in driver MBean: JPPFNodeForwardingMBean, which provides methods to invoke methods, or get or set attributes on remote node MBeans. Each of its methods requires a NodeSelector argument and an MBean name, to determine to which nodes, and which MBean in these nodes, the request will be performed. The return value is always a map of node UUIDs to the corresponding value returned by the request (if any) to the corresponding node. If an exception is raised when performing the request on a specific node, then that exception is returned in the map. Here is an example:

```
JPPFClient client = ...;
AbstractJPPFClientConnection conn =
  (AbstractJPPFClientConnection) client.getClientConnection();
JMXDriverConnectionWrapper driverJmx = conn.getJmxConnection();

JPPFNodeForwardingMBean proxy = driverJmx.getProxy(
  JPPFNodeForwardingMBean.MBEAN_NAME, JPPFNodeForwardingMBean.class);

// this selector selects all nodes attached to the driver
NodeSelector selector = new NodeSelector.AllNodes();
// this selector selects all nodes that have more than 2 processors
ExecutionPolicy policy = new MoreThan("availableProcessors ", 2);
NodeSelector selector2 = new NodeSelector.ExecutionPolicySelector(policy);

// invoke the state() method on the remote 'JPPFNodeAdminMBean' node MBeans
// note that the MBean name does not need to be stated explicitly
Map<String, Object> results = proxy.state(selector);
// this is an exact equivalent, explicitly stating the target MBean on the nodes:
String targetMBeanName = JPPFNodeAdminMBean.MBEAN_NAME;
Map<String, Object> results2 = proxy.forwardInvoke(selector, targetMBeanName, "state");
// handling the results
for (Map.Entry<String, Object> entry: results) {
  if (entry.getValue() instanceof Exception) {
    // handle the exception ...
  } else {
    JPPFNodeState state = (JPPFNodeState) entry.getValue();
    // handle the result ...
  }
}
```

6.3.3 Forwarding JMX notifications

JPPF provides a way to subscribe to notifications from a set of selected nodes, which differs from the one specified in the JMX API. This is due to the fact that the server-side mechanism for the registration of notification listeners is unspecified and thus provides no reliable way to override it.

To circumvent this difficulty, the registration of the notification listener is performed via the JMX client wrapper JMXDriverConnectionWrapper:

- to add a notification listener, use registerForwardingNotificationListener(NodeSelector selector, String mBeanName, NotificationListener listener, NotificationFilter filter, Object handback). This will register a notification listener for the specified MBean on each of the selected nodes. This method returns a listener ID which will be used to remove the notification listener later on. Thus, the application must be careful to keep track of all registered listener IDs.
- to remove a notification listener, use unregisterForwardingNotificationListener(String listenerID).

The notifications forwarded from the nodes are all wrapped into instances of JPPFNodeForwardingNotification. This class, which inherits from Notification, provides additional APIs to identify from which node and which MBean the notification was emitted.

The following code sample puts it all together:

```
JPPFClient client = ...;
AbstractJPPFClientConnection conn =
  (AbstractJPPFClientConnection) client.getClientConnection();
JMXDriverConnectionWrapper driverJmx = conn.getJmxConnection();

// this selector selects all nodes attached to the driver
NodeSelector selector = new NodeSelector.AllNodes();

// create a JMX notification listener
NotificationListener myListener = new NotificationListener() {
  @Override
  public void handleNotification(Notification notification, Object handback) {
    JPPFNodeForwardingNotification wrapping =
      (JPPFNodeForwardingNotification) notification;
    System.out.println("received notification from nodeUuid=" + wrapping.getNodeUuid()
      + ", mBeanName=" + wrapping.getMBeanName());
    // get the actual notification sent by the node
    TaskExecutionNotification actualNotif =
      (TaskExecutionNotification) wrapping.getNotification();
    // handle the notification data ...
  }
}

// register the notification listener with the JPPFNodeTaskMonitorMBean
// on the selected nodes
String listenerID = driverJmx.registerForwardingNotificationListener(
  selector, JPPFNodeTaskMonitorMBean.MBEAN_NAME, listener, null, null);

// ... submit a JPPF job ...

// once the job has completed, unregister the notification listener
driverJmx.unregisterForwardingNotificationListener(listenerID);
```

6.4 JVM health monitoring

The JPPF management APIs provide some basic abilties to monitor the JVM of remote nodes and drivers in a Grid.

These capabilities include:

- memory usage (heap and non-heap)
- CPU load
- count of live threads and deadlock detection
- triggering remote thread dumps and displaying them locally
- triggering remote garbage collections
- triggering remote heap dumps

These features are available via the built-in MBean interface DiagnosticsMBean, defined as follows:

```java
public interface DiagnosticsMBean {
  // The name of this MBean in a driver
  String MBEAN_NAME_DRIVER = "org.jppf:name=diagnostics,type=driver";

  // The name of this MBean in a node
  String MBEAN_NAME_NODE = "org.jppf:name=diagnostics,type=node";

  // Get the memory usage info for the whole JVM
  MemoryInformation memoryInformation() throws Exception;

  // Perform a garbage collection, equivalent to System.gc()
  void gc() throws Exception;

  // Get a full thread dump, including detection of deadlocks
  ThreadDump threadDump() throws Exception;

  // Determine whether a deadlock is detected in the JVM
  Boolean hasDeadlock() throws Exception;

  // Get a summarized snapshot of the JVM health
  HealthSnapshot healthSnapshot() throws Exception;

  // Trigger a heap dump of the JVM
  String heapDump() throws Exception;

  // Get an approximation of the current CPU load
  Double cpuLoad();
}
```

Example usage:

```java
// connect to the driver's JMX server
JMXDriverConnectionWrapper jmxDriver =
  new JMXDriverConnectionWrapper(driverHost, driverPort, false);
// obtain a proxy to the diagnostics MBean
DiagnosticsMBean driverProxy =
  jmxDriver.getProxy(DiagnosticsMBean.MBEAN_NAME_DRIVER, DiagnosticsMBean.class);
// get a thread dump of the remote JVM
ThreadDump tdump = proxy.threadDump();
// format the thread dump as easily readable text
String s = TextThreadDumpWriter.printToString(tdump, "driver thread dump");
System.out.println(s);
```

The MBean interface is exactly the same for nodes and drivers, only the MBean name varies. Also note that, as for other node-related MBeans, it can be be invoked via the node forwarding MBean instead of directly.

Also please keep in mind that, as with all JPPF built-in MBeans, this one is defined as a pluggable MBean, as if it were a custom one.

6.4.1 Memory usage

A detailed memory usage can be obtained by calling the method DiagnosticsMBean.memoryInformation(). This method returns an instance of MemoryInformation, defined as follows:

```
public class MemoryInformation implements Serializable {
  // Get the heap memory usage
  public MemoryUsageInformation getHeapMemoryUsage()

  // Get the non-heap memory usage
  public MemoryUsageInformation getNonHeapMemoryUsage()
}
```

Both heap and non-heap usage are provided as instances of the class MemoryUsageInformation:

```
public class MemoryUsageInformation implements Serializable {
  // Get the initial memory size
  public long getInit()

  // Get the current memory size
  public long getCommitted()

  // Get the used memory size
  public long getUsed()

  // Get the maximum memory size
  public long getMax()

  // Return the ratio of used memory over max available memory
  public double getUsedRatio()
}
```

6.4.2 Thread dumps

You can trigger and obtain a full thread dump of the remote JVM by calling DiagnosticsMBean.threadDump(). This method returns an instance of ThreadDump. Rather than detailing this class, we invite you to explore the related Javadoc, Instead, we would like to talk about the provided facilities to translate thread dumps into readable formats..

There are two classes that you can use to print a heap dump to a character stream:

- TextThreadDumpWriter prints a thread dump to a plain text stream
- HTMLThreadDumpWriter prints a thread dump to a styled HTML stream

Both implement the ThreadDumpWriter interface, defined as follows:

```
public interface ThreadDumpWriter extends Closeable {

  // Print the specified string without line terminator
  void printString(String message);

  // Print the deadlocked threads information
  void printDeadlocks(ThreadDump threadDump);

  // Print information about a thread
  void printThread(ThreadInformation threadInformation);

  // Print the specified thread dump
  void printThreadDump(ThreadDump threadDump);
}
```
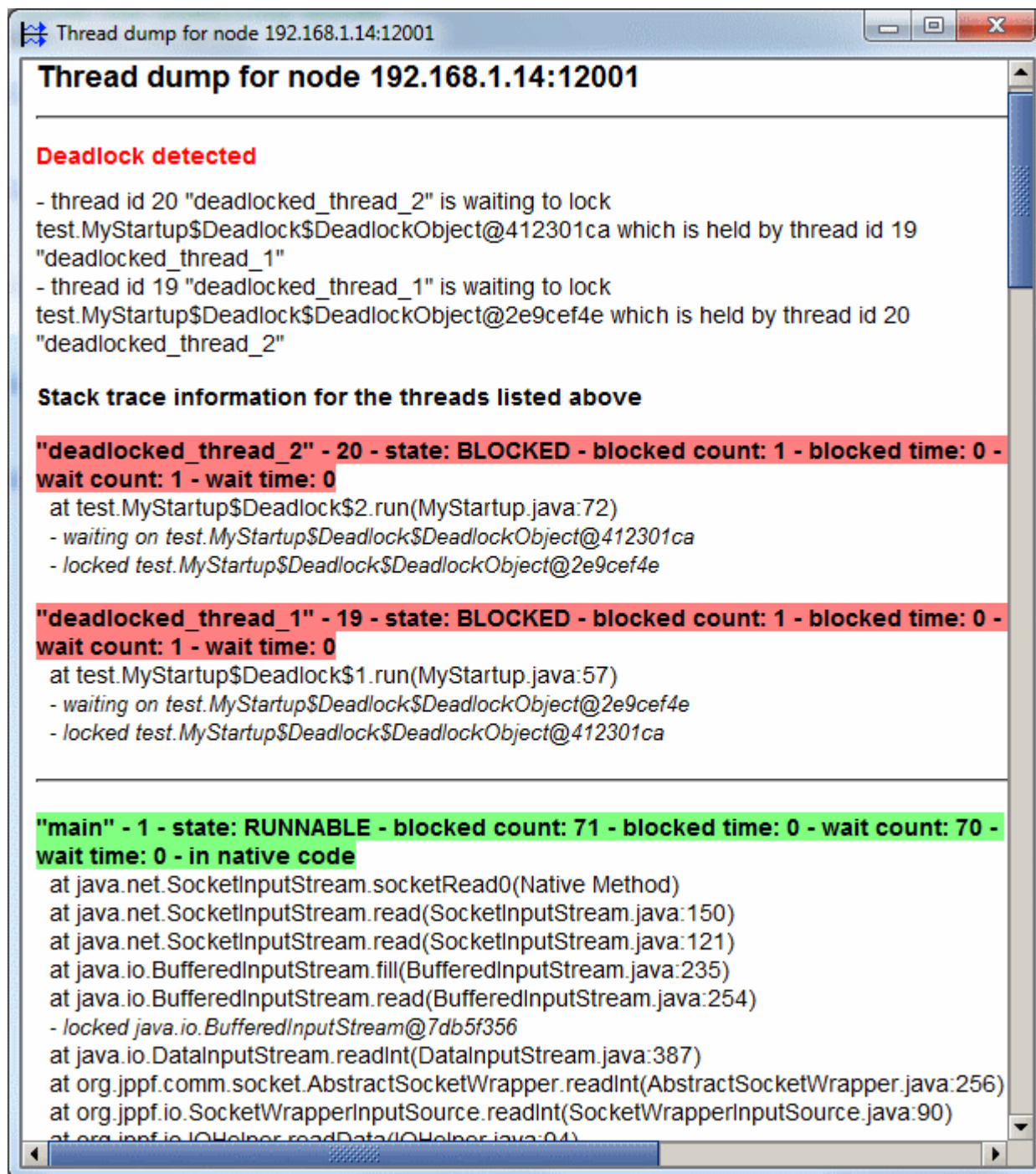
Each of these classes provides a static method to print a thread dump directly into a String:

- static String TextThreadDumpWriter.printToString(ThreadDump tdump, String title)
- static String HTMLThreadDumpWriter.printToString(ThreadDump tdump, String title)

Example usage:

```
// get a thread dump from a remote node or driver
DiagnosticsMBean proxy = ...;
ThreadDump tdump = proxy.threadDump();
// we will print it to an HTML file
FileWriter fileWriter = new FileWriter("MyThreadDump.html");
HTMLThreadDumpWriter htmlPrinter = new HTMLThreadDumpWriter(fileWriter, "My Title");
htmlPrinter.printTreadDump(tdump);
// close the underlying writer
htmlPrinter.close();
```

Here is an example of the output, as rendered in the JPPF administration console:

Thread dump for node 192.168.1.14:12001

Thread dump for node 192.168.1.14:12001

Deadlock detected

- thread id 20 "deadlocked_thread_2" is waiting to lock test.MyStartup$Deadlock$DeadlockObject@412301ca which is held by thread id 19 "deadlocked_thread_1"
- thread id 19 "deadlocked_thread_1" is waiting to lock test.MyStartup$Deadlock$DeadlockObject@2e9cef4e which is held by thread id 20 "deadlocked_thread_2"

Stack trace information for the threads listed above

"deadlocked_thread_2" - 20 - state: BLOCKED - blocked count: 1 - blocked time: 0 - wait count: 1 - wait time: 0
  at test.MyStartup$Deadlock$2.run(MyStartup.java:72)
  - *waiting on test.MyStartup$Deadlock$DeadlockObject@412301ca*
  - *locked test.MyStartup$Deadlock$DeadlockObject@2e9cef4e*

"deadlocked_thread_1" - 19 - state: BLOCKED - blocked count: 1 - blocked time: 0 - wait count: 1 - wait time: 0
  at test.MyStartup$Deadlock$1.run(MyStartup.java:57)
  - *waiting on test.MyStartup$Deadlock$DeadlockObject@2e9cef4e*
  - *locked test.MyStartup$Deadlock$DeadlockObject@412301ca*

"main" - 1 - state: RUNNABLE - blocked count: 71 - blocked time: 0 - wait count: 70 - wait time: 0 - in native code
  at java.net.SocketInputStream.socketRead0(Native Method)
  at java.net.SocketInputStream.read(SocketInputStream.java:150)
  at java.net.SocketInputStream.read(SocketInputStream.java:121)
  at java.io.BufferedInputStream.fill(BufferedInputStream.java:235)
  at java.io.BufferedInputStream.read(BufferedInputStream.java:254)
  - *locked java.io.BufferedInputStream@7db5f356*
  at java.io.DataInputStream.readInt(DataInputStream.java:387)
  at org.jppf.comm.socket.AbstractSocketWrapper.readInt(AbstractSocketWrapper.java:256)
  at org.jppf.io.SocketWrapperInputSource.readInt(SocketWrapperInputSource.java:90)
  at org.jppf.io.IOHelper.readData(IOHelper.java:94)

6.4.3 Health snapshots

You can obtain a summarized snapshot of the JVM state by calling DiagnosticsMBean.healthSnapshot(), which returns an object of type HealthSnapshot, defined as follows:

```java
public class HealthSnapshot implements Serializable {
  // Get the ratio of used / max for heap memory
  public double getHeapUsedRatio()

  // Get the ratio of used / max for non-heap memory
  public double getNonheapUsedRatio()

  // Determine whether a deadlock was detected
  public boolean isDeadlocked()

  // Get the used heap memory in bytes
  public long getHeapUsed()

  // Get the used non-heap memory in bytes
  public long getNonheapUsed()

  // Get the number of live threads in the JVM
  public int getLiveThreads()

  // Get the cpu load
  public double getCpuLoad()

  // Get this snapshot in an easily readable format, according to the default locale
  public String toFormattedString()

  // Get this snapshot in an easily readable format
  public String toFormattedString(final Locale locale)
}
```

The `toFormattedString()` methods return a nicely formatted string which can be used for debugging or testing purposes. Here is an example output with the "en_US" locale:

```
HealthSnapshot[heapUsedRatio=  15.7 %; heapUsed=  71.7 MB; nonheapUsedRatio=   8.7 %;
nonheapUsed=  11.4 MB; deadlocked=false; liveThreads=90; cpuLoad=  23.6 %]
```

6.4.4 CPU load

The CPU load of a remote JVM can be obtained separately by calling DiagnosticsMBean.cpuLoad(). This method returns an approximation of the latest computed CPU load in the JVM. It is important to understand that the CPU load is not computed each time this method is called. Instead, it is computed at regular intervals, and the latest computed value is returned. The purpose of this is to prevent the computation itself from using up too much CPU time, which would throw off the computed value.

The actual computed value is equal to $SUM_i\{cpuTime(thread_i)\}$ / interval, for all the live threads of the JVM at the time of the computation. Thus, errors may occur, since many threads may have been created then died between two computations. However, in most cases this is a reasonable approximation that does not tax the CPU too heavily.

The interval between computations can be adjusted by setting the following property in a node or driver configuration: jppf.cpu.load.computation.interval = time_in_millis. If unspecified, the default value is 1000 ms.

6.4.5 Deadlock indicator

To determine whether a JVM has deadlocked threads, simply call DiagnosticsMBean.hasDeadlock(). This method is useful if you only need that information, without having to process an entire thread dump, which represents a significant overhead, especially from a network perspective.

6.4.6 Triggering a heap dump

Remotely triggering a JVM heap dump is done by calling DiagnosticsMBean.heapDump(). This method is JVM implementation-dependent, as it relies on non-standard APIs. Thus, it will only work with the Oracle standard and JRockit JVMs, along with the IBM JVM. The returned value is a description of the outcome: it will contain the name of the genreated heap dump file for Oracle JVMs, and a success indicator only for IBM JVMs.

6.4.7 Triggering a garbage collection

A remote garbage collection can be triggered by callling DiagnosticsMBean.gc(). This method simply calls System.gc() and thus has the exact same semantics and constraints.

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`:

```java
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:

```java
// 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:

```java
// 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:

```java
// 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:

```java
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`:

```java
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 <u>JPPFStartup</u>, which itself extends <u>java.lang.Runnable</u>. 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 <u>JPPFNodeStartupSPI</u>, 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:

```
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:

```
public interface JPPFDriverStartupSPI extends JPPFStartup { }
```

As an example, we will create an implementation that simply prints a message when the server starts:

```
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 Alternate object serialization schemes

Throughout its implementation, JPPF performs objects transport and associated serialization by the means of a single interface: JPPFSerialization. By configuring a specific implementation of this interface, you can change the way object serialization and deserialization are performed in a JPPF grid.

Note 1: *when an alternate serialization scheme is specified, it must be used by all JPPF clients, servers and nodes, otherwise JPPF will not work. The implementation class must also be present in the classpath of all JPPF components.*

Note 2: *to the difference of JPPF 3.x and earlier versions, serialization schemes now also apply to objects passed via the management APIs. This implies that the nodes must always have the JMXMP protocol library (jmxremote_optional-1.0_01-ea.jar) in their classpath.*

7.4.1 Implementation

To create a new serialization scheme, you simply need to implement JPPFSerialization, defined as follows:

```
public interface JPPFSerialization {

  // Serialize an object into the specified output stream
  void serialize(Object o, OutputStream os) throws Exception;

  // Deserialize an object from the specified input stream
  Object deserialize(InputStream is) throws Exception;
}
```

For example, we could wrap the default Java serialization into a serialization scheme as follows:

```
public class DefaultJavaSerialization implements JPPFSerialization {
  @Override
  public void serialize(final Object o, final OutputStream os) throws Exception {
    new ObjectOutputStream(os).writeObject(o);
  }

  @Override
  public Object deserialize(final InputStream is) throws Exception {
    return new ObjectInputStream(is).readObject();
  }
}
```

7.4.2 Specifying the JPPFSerialization implementation class

This is done in the JPPF configuration file, by adding this property:

```
# Define the implementation of the JPPF serialization scheme
jppf.object.serialization.class = my_package.MyJPPFSerialization
```

Where *my_package.MyJPPFSerialization* implements the interface JPPFSerialization.

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. The corresponding implementation class is DefaultJavaSerialization.

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 significant 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.serialization.class = org.jppf.serialization.DefaultJPPFSerialization
```

7.4.3.3 XStream-based serialization

JPPF has a built-in Object Stream Builder that uses XStream to provide XML serialization: XstreamSerialization. To use it, simply specify:

```
# configure the object stream builder implementation
jppf.object.stream.builder = org.jppf.serialization.XstreamSerialization
```

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

7.4.3.4 Kryo serialization sample

The Kryo serialization sample provides an implementation of JPPFSerialization which uses the Kryo library for serializing and deserializing Java objects.

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 profile

First let's implement our parameters profile. To this effect, we implement the interface <u>LoadBalancingProfile</u>:

```
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 <u>FixedSizeProfile</u> 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 <u>TypedProperties</u> 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:

```java
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:

```java
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*:

```java
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
  JobMetadata getJobMetadata();

  // Set the current job's metadata
  void setJobMetadata(JobMetadata metadata);
}
```

When implementing this interface, the job metadata becomes accessible via an instance of JobMetadata.

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 node connection events in the server

This extension point allows you to register a listener for receiving notifications when a node is connected to, or disconnected from the server. As for other JPPF extensions, it relies on the Service Provider Interface (SPI) mechanism to enable an easy registration.

To implement this extension, you first need to create an implementation of the NodeConnectionListener interface, defined as follows:

```
public interface NodeConnectionListener extends EventListener {
   // Called when a node is connected to the server
   void nodeConnected(NodeConnectionEvent event);

   // Called when a node is disconnected from the server
   void nodeDisconnected(NodeConnectionEvent event);
}
```

Each notification method receives instances of the NodeConnectionEvent class, which is defined as:

```
public class NodeConnectionEvent extends EventObject {
   // Get the node information for this event
   public JPPFManagementInfo getNodeInformation()
}
```

As we can see, these event objects are simple wrappers carrying detailed information about the node, via the class JPPFManagementInfo :

```
public class JPPFManagementInfo
   implements Serializable, Comparable<JPPFManagementInfo> {

   // Get the host on which the node is running
   public String getHost()

   // Get the port on which the node's JMX server is listening
   public int getPort()

   // Get the system information associated with the node at the time
   // it established the connection
   public JPPFSystemInformation getSystemInfo()

   // Get the node's unique id (UUID)
   public String getId()

   // Determine whether this information represents another driver,
   // connected as a peer to the current driver
   public boolean isDriver()

   // Determine whether this information represents a real node
   public boolean isNode()
}
```

For details on the available information, we encourage you to read the Javadoc for the class JPPFSystemInformation.

Note: *from the nodeConnected() method, you may refuse the connection by throwing a RuntimeException. This will cause the JPPF driver to terminate the connection.*

To deploy the extension:

– create a file named **org.jppf.server.event.NodeConnectionListener** in the **META-INF/services** folder
– in this same file, add the fully qualified class name of your NodeConnectionListener implementation, for example:
  mypackage.MyNodeConnectionListener. This is the service definition file for the extension.
– create a jar with your code and and service definition file and add it to the driver's classpath, or simply add your classes folder to the driver's classpath.

7.7 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.

7.7.1 NodeLifeCycleListener interface

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 when the node has loaded a job header and before
  // the DataProvider or any of the tasks has been loaded
  void jobHeaderLoaded(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 object representing the current JPPF node
  public Node getNode()
  // The type of this event
  public NodeLifeCycleEventType getType()
  // Get the job currently being executed
  public JPPFDistributedJob getJob();
  // Get the tasks currently being executed
  public List<Task> getTasks();
  // Get the data provider for the job
  public DataProvider getDataProvider();
  // Get the class loader used to load the tasks and
  // the classes they need from the client
  public AbstractJPPFClassLoader getTaskClassLoader()
}
```

Please note that the methods getJob(), getTasks() and getTaskClassLoader() 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. The type of the event is available as an instance of the typesafe enum NodeLifeCycleEventType, defined as follows:

```
public enum NodeLifeCycleEventType {
  // nodeStarting() notification
  NODE_STARTING,
  // nodeEnding() notification
  NODE_ENDING,
  // jobHeaderLoaded() notification
  JOB_HEADER_LOADED,
  // jobStarting() notification
  JOB_STARTING,
  // jobEnded() notification
  JOB_ENDING
 }
```

You will also notice that the method getTasks() returns a list of Task<T> instances. Task<T> is the interface for all JPPF tasks, and can be safely cast to JPPFTask for all practical purposes.

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 getName();

  // Get the universal unique id for this job
  String getUuid();

  // Get the service level agreement between the job and the server
  JobSLA getSLA();

  // Get the user-defined metadata associated with this job
  JobMetadata getMetadata();
}
```

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 either the JPPF driver's class path, if you want it deployed to all nodes connected to that driver, or to the classpath of individual nodes, if you only wish specific nodes to have the add-on.

Note regarding the `jobHeaderLoaded()` notification:
At the time this method is called, neither the DataProvider (if any) nor the tasks have been deserialized. This means that the tasks can reference classes that are not yet in the classpath, and you can add these classes to the classpath on the fly, for instance by calling NodeLifeCycleEvent.getTaskClasLoader(), then invoking the addURL(URL) method of the resulting AbstractJPPFClassLoader.

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 {
  @Override
  public void nodeStarting(NodeLifeCycleEvent event) {
    System.out.println("node ready to process jobs");
  }

  @Override
  public void nodeEnding(NodeLifeCycleEvent event) {
    System.out.println("node ending");
  }

  @Override
  public void jobHeaderLoaded(NodeLifeCycleEvent event) {
    JPPFDistributedJob job = event.getJob();
    System.out.println("node loaded header for job '" + job.getId() +
      "' using task class loader " + event.getTaskClassLoader());
  }

  @Override
  public void jobStarting(NodeLifeCycleEvent event) {
    JPPFDistributedJob job = event.getJob();
    System.out.println("node starting job '" + job.getId() + "' with " +
      event.getTasks().size() + " tasks");
  }

  @Override
  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 samples:
– "NodeLifeCycle"
– "Node Tray"
– "Extended Class Loading"

7.7.2 Error handler

It is now possible to provide an error handler for each `NodeLifeCycleListener` implementation. This error handler will process all uncaught `Throwable`s raised during the execution of any of the listener's methods.

To setup an error handler on a `NodeLifeCycleListener` implementation, you just have to implement the interface NodeLifeCycleErrorHandler, defined as follows:

```
public interface NodeLifeCycleErrorHandler {
  // Handle the throwable raised for the specified event
  void handleError(NodeLifeCycleListener listener, NodeLifeCycleEvent event,
                   Throwable t);
}
```

The `listener` parameter is the listener instance which threw the throwable. The `event` parameter is the notification that was sent to the listener; its `getType()` method will allow you to determine which method of the listener was called when the throwable was raised. The last parameter is the actual throwable that was raised.

When the NodeLifeCycleListener does not implement `NodeLifeCycleErrorHandler`, JPPF will delegate the error handling to a default implementation: DefaultLifeCycleErrorHandler.

Lastly, if an uncaught throwable is raised within the error handler itself, JPPF will handle it as follows:

- if the logging level is "debug" or finer then the full stack trace of the throwable is logged
- otherwise, only the throwable's class and message are logged
- if the throwable is an instance of Error, it is propagated up the call stack

7.8 Node initialization hooks

In the JPPF nodes, the lookup for a server to connect to relies essentially on each node's configuration. Thus, to implement a customized server lookup or failover mechanism, it is necessary to be able to modify the configuration, before the server lookup and connection is attempted. To this effect, JPPF provides a pluggable initialization hook which can be executed by the node before each connection attempt.

An initialization hook is a Java class that implements the interface <u>InitializationHook</u>, which is defined as follows:

```
public interface InitializationHook {
  // Called each time the node is about to attempt to connect to a driver
  void initializing(UnmodifiableTypedProperties initialConfiguration);
}
```

Note that the `initialConfiguration` parameter reflects the exact same set of configuration properties that were loaded by the node at startup time. It is an instance of <u>UnmodifiableTypedProperties</u>, which is an extension of <u>TypedProperties</u> that does not permit the modification, insertion or removal of any property. To modify the node's configuration, you have to use <u>JPPFConfiguration.getProperties()</u>, which reflects the current configuration and can be modified.

Here is an example implementation:

```
public class MyInitializationHook extends InitializationHook {
  // an alternate server address read from the configuration
  private String alternateServer = null;
  // determines which server address to use
  private boolean useAlternate = false;

  // This method toggles the JPPF server address between the value set in the
  // configuration file and an alternate server address
  public void initializing(UnmodifiableTypedProperties initialConfiguration) {
    // store the alternate server address
    if (alternateServer == null) {
      alternateServer = initialConfiguration.getString("alternate.server.host");
    }
    TypedProperties currentConfig = JPPFConfiguration.getProperties();
    // means the JPPF-configured value is to be used
    if (!useAlternate) {
      // reset the server address to its initially configured value
      String initialServer = initialConfiguration.getString("jppf.server.host");
      currentConfig.setProperty("jppf.server.host", initialServer);
      // toggle the server address to use for the next attempt
      useAlternate = true;
    } else {
      // connection to JPPF-configured server failed,
      // we will now try to connect to the alternate server
      currentConfig.setProperty("jppf.server.host", alternateServer);
      // toggle the server address to use for the next attempt
      useAlternate = false;
    }
  }
}
```

Once the implementation is done, the initialization hook is plugged into JPPF using the service provider interface:

– create a file in **META-INF/services** named "**org.jppf.node.initialization.InitializationHook**"
– 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 classpath of each node.

Related sample: Initialization Hook sample.

7.9 Fork/Join thread pool in the nodes

By default, JPPF nodes use a "standard" thread pool for executing tasks. This add-on allows the use of a fork/join thread pool instead of the standard one. This enables JPPF tasks to locally (in the node) spawn `ForkJoinTask` (or any of its subclasses) instances and have them processed as expected for a `ForkJoinPool`.

To use this add-on, you will need to deploy the jar file "ThreadManagerForkJoin.jar" to either the JPPF server's or node's classpath. If deployed in the server's classpath, it will be available to all nodes.

The next step is to configure each node for use of the fork/join thread pool. This is achieved by adding the following property to the node's configuration:

```
jppf.thread.manager.class = org.jppf.server.node.fj.ThreadManagerForkJoin
```

Here is an example usage, which computes the number of occurrences of each word in a set of documents:

```java
public class WordCountTask extends JPPFTask {
  // a list of docuuments to process
  private final List<String> documents;

  public WordCountTask(final List<String> documents) {
    this.documents = documents;
  }

  @Override
  public void run()
  {
    List<Map<String, Integer>> results = new ArrayList<>();
    // compute word counts in each document
    if (ForkJoinTask.inForkJoinPool()) {
      List<ForkJoinTask<Map<String, Integer>>> tasks = new ArrayList<>();
      // fork one new task per document
      for (String doc: documents) tasks.add(new MyForkJoinTask(doc).fork());
      // wait until all forked tasks have completed (i.e. join)
      for (ForkJoinTask<Map<String, Integer>> task: tasks) results.add(task.join());
    } else {
      // if not in FJ pool, process documents sequentially
      for (String doc: documents) results.add(new MyForkJoinTask(doc).compute());
    }
    // merge the results of all documents
    Map<String, Integer> globalResult = new HashMap<>();
    for (Map<String, Integer> map: results) {
      for (Map.Entry<String, Integer> entry: map.entrySet()) {
        Integer n = globalResult.get(entry.getKey());
        if (n == null) globalResult.put(entry.getKey(), entry.getValue());
        else globalResult.put(entry.getKey(), n + entry.getValue());
      }
    }
    // set the merged word counts as this task's result
    this.setResult(globalResult);
  }
}
```

We can see here that the execution strategy depends on the result of calling `ForkJoinTask.inForkJoinPool()`: if we determine that a fork/join pool is available, then a new task is forked for each document, and thus executed asynchronously. The execution is then synchronized by joining each forked task. Otherwise, the documents are processed sequentially.

In this example, our fork/join task is defined as follows:

```java
public class MyForkJoinTask extends RecursiveTask<Map<String, Integer>> {
  // remove spaces and non-word characters
  private static Pattern pattern = Pattern.compile("\\s|\\W");
  private final String document;

  public MyForkJoinTask(final String document) {
    this.document = document;
  }

  @Override
  // return a mapping of each word to its number of occurrences
  public Map<String, Integer> compute() {
    Map<String, Integer> result = new HashMap<>();
    // split the document into individual words
    String[] words = pattern.split(document);
    // count the number of occurrences of each word in the document
    for (String word: words) {
      Integer n = result.get(w);
      result.put(word, (n == null) ? 1 : n+1);
    }
    return result;
  }
}
```

Related sample: the fork/join thread pool add-on of the JPPF distribution provides a more sophisticated example, taking full advantage of the fork/join features in Java 7. This example is packaged along with the downloadable "JPPF-x.y.z-jdk7-addons.zip" file.

7.10 Receiving notifications of class loader events

It is possible to receive notifications of whether a class was loaded or not found by a JPPF class loader. This can be done by implementing the interface ClassLoaderListener defined as follows:

```
public interface ClassLoaderListener extends EventListener {
  // Called when a class has been successfully loaded by a class loader
  void classLoaded(ClassLoaderEvent event);

  // Called when a class was not found by a class loader
  void classNotFound(ClassLoaderEvent event);
}
```

Each notification provides an event object which is an instance of the class ClassLoaderEvent, defined as:

```
public class ClassLoaderEvent extends EventObject {
  // Get the class that was successfully loaded or null if the class was not found
  public Class<?> getLoadedClass()
  // Get the name of the class that was loaded or not found
  public String getClassName()
  // Determine whether the class was loaded from the class loader's URL classpath
  // If false, then the class was loaded from a remote JPPF driver or client
  public boolean isFoundInURLClassPath()
  // Get the class loader which emitted this event
  public AbstractJPPFClassLoader getClassLoader()
}
```

Note that you may receive up to two notfications for the same class, due to the parent delegation model in the JPPF class loader hierarchy: when a node attempts to load a class from a client class loader (i.e. which accesses the classpath of a remote JPPF client), it will first delegate to its parent class loader, which is a driver class loader. As a result, the parent class loader will send a "classNotFound()" notification, and then the client class loader will send a second notification after it attempts to load the class from the client's classpath. Please refer to the Class Loading in JPPF documentation for full details on how class loading works.

Once the implementation is done, the class loader listener is plugged into JPPF using the service provider interface:
– create a file in **META-INF/services** named "**org.jppf.node.classsloader.ClassLoaderListener**"
– 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 classpath of each node.

As an example, let's say we simply want to print the notifications received by a listener, which we implement as follows:

```
package test;
import org.jppf.classloader.*;

public class MyClassLoaderListener implements ClassLoaderListener {
  @Override
  public void classLoaded(final ClassLoaderEvent event) {
    AbstractJPPFClassLoader cl = event.getClassLoader();
    System.out.println("loaded " + event.getLoadedClass() + " from "
      + (cl.isClientClassLoader() ? "client" : "server") + " class loader in "
      + (event.isFoundInURLClasspath() ? "local" : "remote") + " classpath");
  }

  @Override
  public void classNotFound(final ClassLoaderEvent event) {
    AbstractJPPFClassLoader cl = event.getClassLoader();
    System.out.println("class " + event.getClassName() + " was not found by "
      + (cl.isClientClassLoader() ? "client" : "server") + " class loader");
  }
}
```

Then we create the file "META-INF/services/org.jppf.node.classsloader.ClassLoaderListener" with this content:
```
test.MyClassLoaderListener
```

All that remains to do is to package the class and service files into a jar and add this jar to the classpath of the nodes.

7.11 Receiving notifications from the tasks

We have seen in "*Development guide > Task objects > Sending notifications from a task*" that JPPF tasks can send notifications with the method Task.fireNotification(Object, boolean). It is possible to register listeners for these notifications via the the service provider interface (SPI). These listeners must implement the interface TaskExecutionListener, defined as follows:

```
public interface TaskExecutionListener extends EventListener {
  // Called by the JPPF node to notify a listener that a task was executed
  void taskExecuted(TaskExecutionEvent event);

  // Called when a task sends a notification via Task.fireNotfication(Object, boolean)
  void taskNotification(TaskExecutionEvent event);
}
```

The method `taskExecuted()` is always called by the JPPF node, whereas `taskNotification()` is always invoked by user code. Both methods receive events of type TaskExecutionEvent, defined as follows:

```
public class TaskExecutionEvent extends EventObject {
  // Get the JPPF task from which the event originates
  public Task<?> getTask()

  // Get the object encapsulating information about the task
  public TaskInformation getTaskInformation()

  // Get the user-defined object to send as part of the notification
  public Object getUserObject()

  // If true then also send this notification via the JMX Mbean,
  // otherwise only send to local listeners
  public boolean isSendViaJmx()
}
```

Once a task execution listener is implemented, it is plugged into the JPPF nodes using the service provider interface:
– create a file in **META-INF/services** named "**org.jppf.node.event.TaskExecutionListener**"
– 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 classpath of the server, if you want all the nodes to register a listner instance, or to the classpath of specific individual nodes.

The following example prints the notifications it receives to the node's output console:

```
package my.test;
import ...;
public class MyTaskListener implements TaskExecutionListener {
  @Override public synchronized void taskExecuted(TaskExecutionEvent event) {
    System.out.println("Task " + event.getId() + " completed with result : " +
      event.getTask().getResult());
    System.out.println("cpu time = " + event.getTaskInformation().getCpuTime() +
      ", elapsed = " + event.getTaskInformation().getElapsedTime());
  }

  @Override public synchronized void taskNotification(TaskExecutionEvent event) {
    System.out.println("Task " + event.getId() + " sent user object : " +
      event.getUserObject());
  }
}
```

To use this listener, you need to create a file "META-INF/services/org.jppf.node.event.TaskExecutionListener" containing the line "**my.test.MyTaskListener**" and add it, along with the implementation class, to the server's or node's classpath.

7.12 JPPF node screensaver

A screensaver can be associated with a running JPPF node. The screensaver is a Java Swing UI which is displayed at the time the node is launched. It can run in full screen or in windowed mode, depending on the configuration settings.

7.12.1 Creating a custom screensaver

A screensaver implements the interface JPPFScreenSaver, defined as follows:

```
public interface JPPFScreenSaver {
  // Get the Swing component for this screen saver
  JComponent getComponent();

  // Initialize this screen saver, and in particular its UI components
  void init(TypedProperties config, boolean fullscreen);

  // Destroy this screen saver and release its resources
  void destroy();
}
```

The screensaver lifecycle is as follows:

- the screensaver is instantiated, based on a class name specified in the configuration (requires a no-arg constructor)
- the `init()` method is called, passing in the JPPF configuration and a flag indicating whether the full screen mode is both requested and supported
- a `JFrame` is created and the component obtained by calling `getComponent()` is added to this `JFrame`. In full screen, the frame is undecorated (no caption, menu bar, status bar or borders) and will cover all available space on all available monitors: the screen saver will spread over all screens in a multi-monitor setup
- the frame is then made visible
- finally, the `destroy()` method is called when the frame is closed. In full screen mode, this happens upon pressing a key or clicking or (optionally) moving the mouse

The code which handles the screensaver is implemented as a node initialization hook: this means it starts just after the configuration has been read.

Here is a sample screensaver implementation which draws a number of small circles at random locations and with a random color, around 25 times per second. Addtionally, the screen is cleared every 5 seconds and the process starts over:

```
public class SimpleScreenSaver extends JPanel implements JPPFScreenSaver {
  private Random rand = new Random(System.nanoTime());
  private Timer timer = null;
  private volatile boolean reset = false;

  public SimpleScreenSaver() { super(true); }

  @Override
  public JComponent getComponent() {
    return this;
  }

  @Override
  public void init(TypedProperties config, boolean fullscreen) {
    setBackground(Color.BLACK);
    timer = new Timer("JPPFScreenSaverTimer");
    timer.scheduleAtFixedRate(new TimerTask() {
      @Override public void run() { repaint(); }
    }, 40L, 40L); // draw new circles every 40 ms or 25 times/second
    timer.scheduleAtFixedRate(new TimerTask() {
      @Override public void run() { reset = true; }
    }, 5000L, 5000L); // clear the screen every 5 seconds
  }

  @Override
  public void destroy() {
    if (timer != null) timer.cancel();
  }
```

```
  @Override
  public void paintComponent(final Graphics g) {
    int w = getWidth();
    int h = getHeight();
    if (reset) { // if reset requested, clear the screen
      reset = false;
      g.setColor(Color.BLACK);
      g.fillRect(0, 0, w, h);
    } else { // draw 500 small circles
      int n = 5;
      for (int i=0; i<500; i++) {
        // random x, y coordinates
        int x = rand.nextInt(w-(n-1));
        int y = rand.nextInt(h-(n-1));
        // random color
        g.setColor(new Color(rand.nextInt(256), rand.nextInt(256), rand.nextInt(256)));
        g.fillOval(x, y, n, n);
      }
    }
  }
}
```

It will look like this:
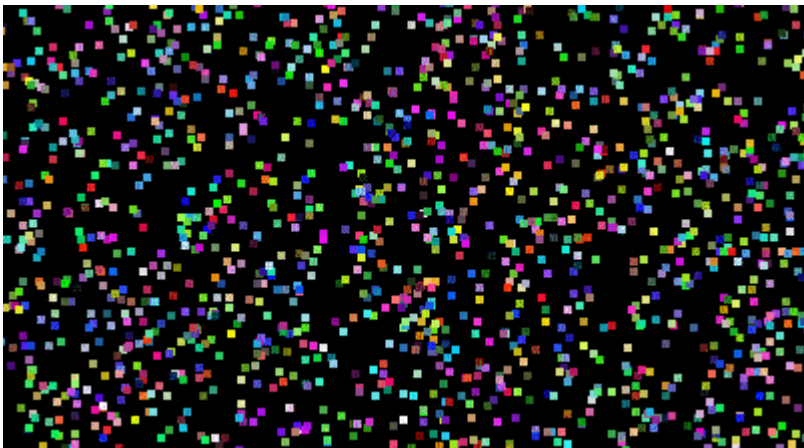
7.12.2 Integrating with node events

To receive notifications of node lifce cycle events and/or individual tasks completion, you will need to implement the NodeIntegration interface or, more conveniently, extend the adapter class NodeIntegrationAdapter. NodeIntegration is defined as follows:

```
public interface NodeIntegration extends NodeLifeCycleListener, TaskExecutionListener {
  // Provide a reference to the screen saver
  void setScreenSaver(JPPFScreenSaver screensaver);
}
```

As we can see, this is just an interface which joins both NodeLifeCycleListener and TaskExecutionListener and provides a way to hook up with the screensaver.

The following example shows how to use node events to display and update the number of tasks executed by the node in the screensaver:

```java
// displays the number of executed tasks in a JLabel
public class MyScreenSaver extends JPanel implements JPPFScreenSaver {
  private JLabel nbTasksLabel = new JLabel("number of tasks: 0");
  private int nbTasks = 0;

  @Override public JComponent getComponent() { return this; }
  @Override public void init(TypedProperties config, boolean fullscreen) {
    this.add(nbTasksLabel);
  }
  @Override public void destroy() { }

  public void updateNbTasks(int n) {
    nbTasks += n;
    nbTasksLabel.setText("number of tasks: " + nbTasks);
  }
}

// update the number of tasks on each job completion event
public class MyNodeIntegration extends NodeIntegrationAdpater {
  private MyScreenSaver screensaver = null;

  @Override public void jobEnding(NodeLifeCycleEvent event) {
    if (screensaver != null) screensaver.updateNbTasks(event.getTasks().size());
  }

  @Override public void setScreenSaver(JPPFScreenSaver screensaver) {
    this.screensaver = (MyScreenSaver) sceensaver;
  }
}
```

7.12.3 Configuration

The screensaver supports a number of configuration properties which allow a high level of customization:

```
# enable/disable the screen saver, defaults to false (disabled)
jppf.screensaver.enabled = true

# the screensaver implementation: fully qualified class name of an implementation of
# org.jppf.node.screensaver.JPPFScreenSaver
jppf.screensaver.class = org.jppf.node.screensaver.impl.JPPFScreenSaverImpl

# the node event listener implementation: fully qualified class name of an
# implementation of org.jppf.node.screensaver.NodeIntegration
# if left unspecified or empty, no listener will be used
jppf.screensaver.node.listener = org.jppf.node.screensaver.impl.NodeState

# title of the JFrame used in windowed mode
jppf.screensaver.title = JPPF is cool

# path to the image for the frame's icon (in windowed mode)
jppf.screensaver.icon = org/jppf/node/jppf-icon.gif

# display the screen saver in full screen mode?
# in full screen mode, the screen saver will take all available screen space on all
# available monitors, the mouse cursor is not displayed, and the node will exit on any
# key pressed, mouse click or mouse motion
jppf.screensaver.fullscreen = true

# width and height (in pixels), only apply if fullscreen = false. Default to 1000x800
jppf.screensaver.width = 1000
jppf.screensaver.height = 800

# close on mouse motion in full screen mode? default to true
jppf.screensaver.mouse.motion.close = true
```

7.12.4 JPPF built-in screensaver

A built-in screensaver is provided, which displays an number of moving small logos (images) bouncing around the screen and bouncing against each other. Additionally, it displays panel comprising a (personalizable) logo, along with a status bar indicating how long the node has been running, its connection status, execution status and the number of tasks it has executed. It looks like this:
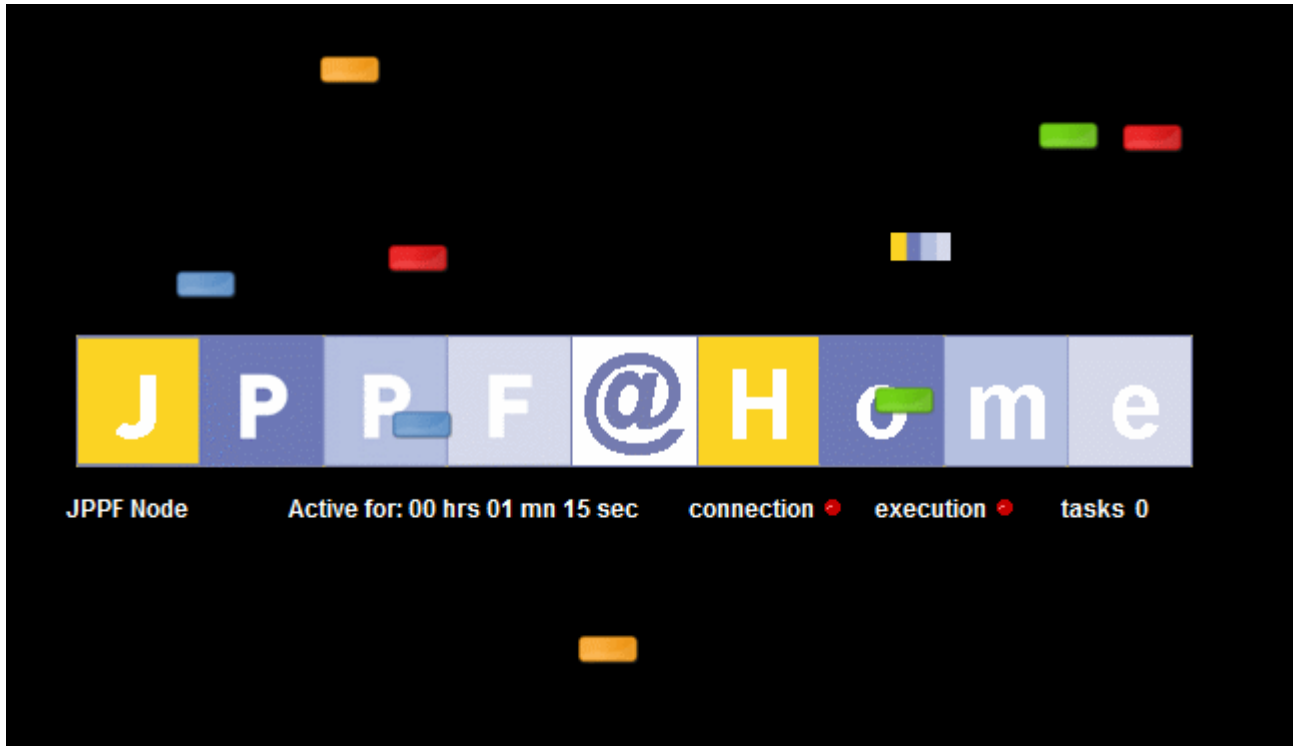
The built-in screensaver proposes a number of spceific configuration options, which aim at providing a fine level of personalization and hopefully some fun!

```
# should collisions between moving logos be handled? defaults to true
jppf.screensaver.handle.collisions = true

# number of moving logos
jppf.screensaver.logos = 50

# speed from 1 to 100
jppf.screensaver.speed = 100

# path to the moving logo image(s). Multiple images can be specified, their paths
# must be separated with '|' (pipe) characters. They will be distributed in a
# round-robin fashion according to the number of logos
jppf.screensaver.logo.path = org/jppf/node/jppf_group_small2.gif| \
    org/jppf/node/rectagle_blue.png| \
    org/jppf/node/rectagle_orange.png| \
    org/jppf/node/rectagle_green.png| \
    org/jppf/node/rectagle_red.png

# path to the larger image at the center of the screen
jppf.screensaver.centerimage = org/jppf/node/jppf@home.gif

# horizontal alignment of the status panel (including the larger image).
# useful when using a multi-monitor setup, where a centered panel will be split on two
# screens and thus more difficult to read
# possible values: 'left' or 'l', 'center' or 'c', 'right' or 'r'; default is 'center'
jppf.screensaver.status.panel.alignment = center
```

To use this screensaver, you need to set the following properties:

```
jppf.screensaver.enabled = true
jppf.screensaver.class = org.jppf.node.screensaver.impl.JPPFScreenSaverImpl
jppf.screensaver.node.listener = org.jppf.node.screensaver.impl.NodeState
```

7.13 Flow of customizations in JPPF

The following sections describe the flow of customizations and extensions in JPPF, and especially the order in which they are loaded, both with respect to each other and to the major events in the server and nodes life cycle.

7.13.1 JPPF driver
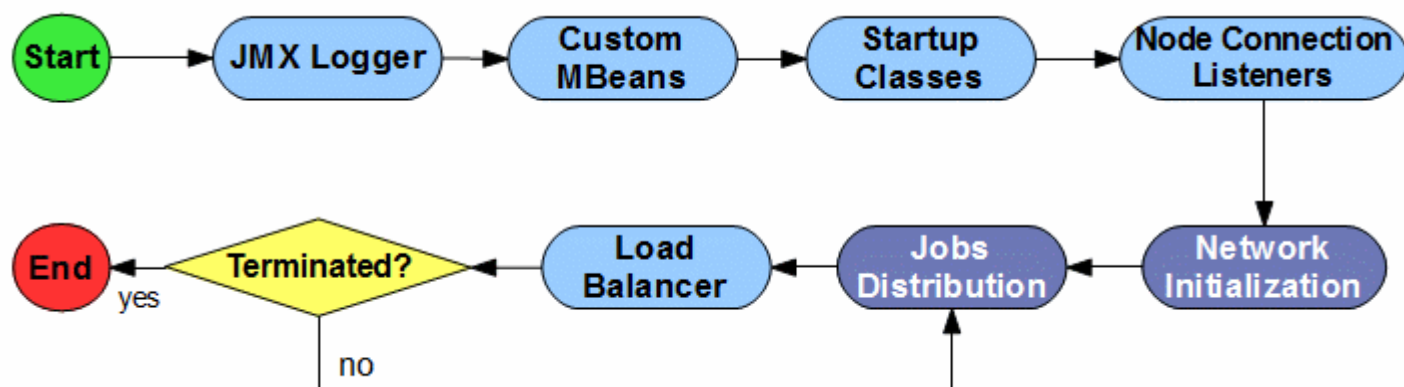
7.13.2 JPPF node
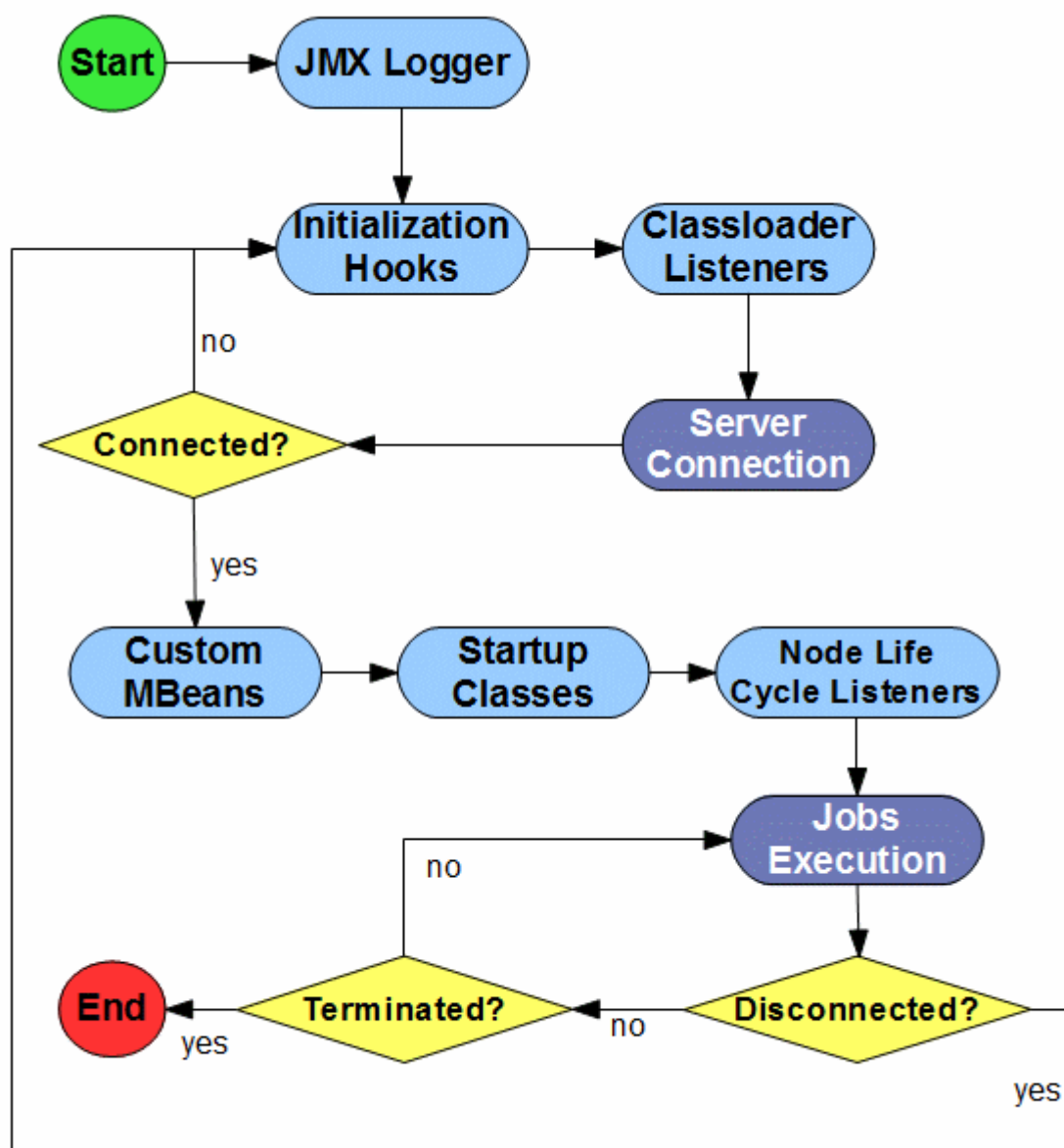
8 Class Loading In JPPF

8.1 How it works

The distributed class loading framework in JPPF is the mechanism that makes it possible to execute code in a node that has not been explicitly deployed to the node's environment. Through this, JPPF tasks whose code (the actual bytecode to execute) is only defined in a JPPF client application, can be executed on remote nodes without the application developer having to worry about how this code will be transported there.

While this mechanism is fully transparent from the client application's perspective, it has a number of implications and particularities that may impact various aspects of JPPF tasks execution, including performance and integration with external libraries.

Let's have a quick view of the path followed by a class loading request at the time a JPPF task is executed within a node:
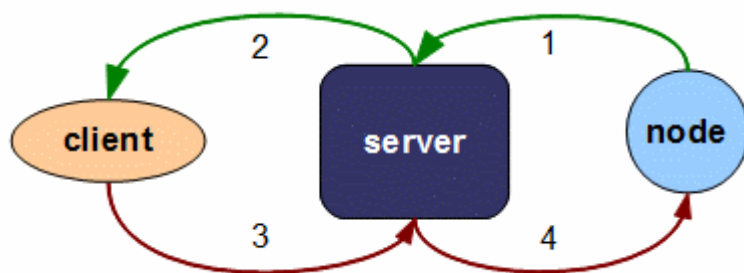
We can see that this class loading request is executed in four steps:

4. the node sends a network request to the remote server for the class
5. the server forwards the request to the identfied remote client
6. the client provides a response (the bytecode of the class) to the server
7. the server forwards the response to the node

Once these steps are performed, the node holds the bytecode of the class and can effectively define and load it as for any standard Java class.

This use case is a simplification of the overall class loading mechanism in JPPF, however it illustrates what actually takes place when a task is executed in a node.This also raises a number of questions that need clarification:

– how does the server know which client to forward a request to?
– how does this fit into the Java class loader delegation model?
– how does it work in complex JPPF topologies with multiple servers?
– how does it apply to JPPF customizations and add-ons or external libraries that are available in the server or node's classpath?
– what is the impact on execution performance?
– what possibilities does this open up for JPPF applications?

We will address these questions in details in the next sections.

8.2 Class loader hierarchy in JPPF nodes

The JPPF class loader mechanism follows a hierarchy based on parent-child relationships between class loader instances, as illustrated in the following picture:
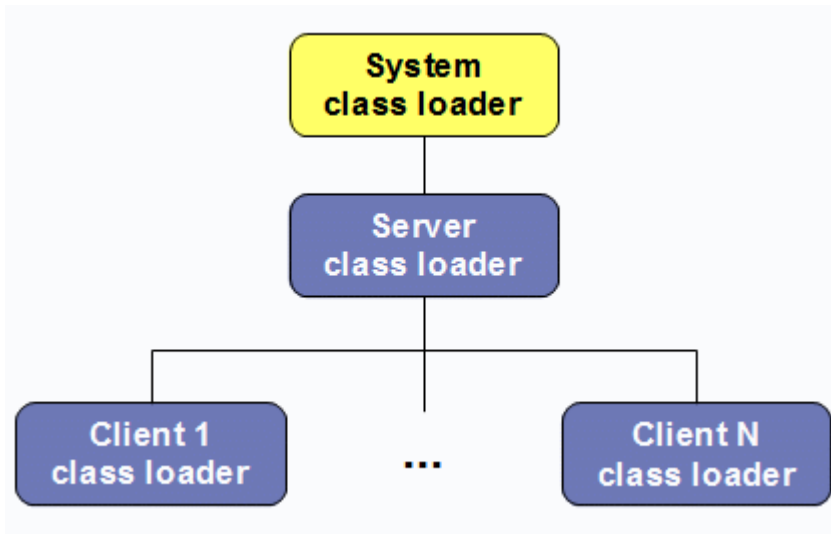
The system class loader is used to start the JPPF node. With most JVMs, it will be an instance of the class `java.net.URLClassloader` and its usage and creation are handled by the JVM.

The server class loader is a concrete implementation of the class `AbstractJPPFClassloader` and provides remote access to classes and resources in the server's classpath. It is created at the time the node establishes a connection to the server. It is also discarded when the node disconnects from the server. The parent of the server class loader is the system class loader. Please note that `AbstractJPPFClassLoader` is also a subclass of `URLClassLoader`.

The client class loaders are also concrete implementations of `AbstractJPPFClassloader` and provide remote access to classes and resources in one or more clients' classpaths. Each client class loader is created the first time the node executes a job which was submitted by that client. Thus, the node may hold many client class loaders.

It is important to note that, by design, the JPPF node holds a single network connection to the server, shared by all instances of `AbstractJPPFClassLoader`, including the server and clients class loaders. This design avoids a lot of potential confusion, inconsistencies and synchronization pitfalls when performing multiple class loading requests in parallel.

By default, a JPPF class loader follows the standard delegation policy to its parent. This means that, when a class is requested from a client class loader, it will first delegate to its parent, the server class loader, who will in turn first delegate to the system class loader. If a class is not found by the parent, then the class loader will look it up in the classpath to which it has access.

Thus, the flow of a request, for a class that is only in a client's class path, becomes a little more complex:
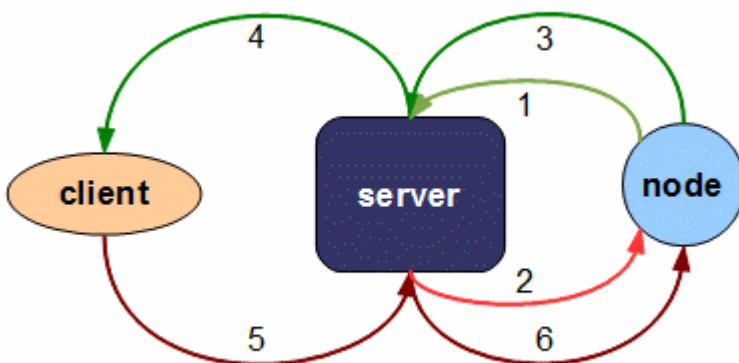
Here, the first two steps are initiated by the server class loader, as a result of the client class loader delegating to its parent. What is missing from this picture are the calls to the system class loader, since they are only meaningful if the requested class is in the node's local classpath.

8.3 Relationship between UUIDs and class loaders

We have seen in the *Development Guide* that each JPPF client has its own identifier, unique across the entire JPPF grid. This is also true of servers and nodes. The client UUID is what allows a JPPF node to know which client a class loader is associated with, and use it to route a class loading request from the node down to the client that submitted the job.

If a node only knows the client UUID, then it will only be able to handle the routing of class loading requests in the simplest JPPF grid topology: a topology which only has one server. However, there is a mechanism that allows the class loading to work in much more complex topologies, such as this one:
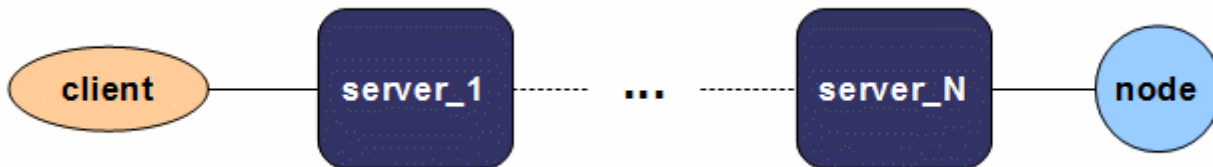
To this effect, each job executed on a node will transport, in addition to the originating client's UUID, the UUID of each server in the chain of servers that had to be traversed to get to the node. In JPPF terminology, this ordered list of UUIDs is called a *UUID path*. With this information known, it is possible to route a class loading request through any chain of servers, as illustrated in the picture below:
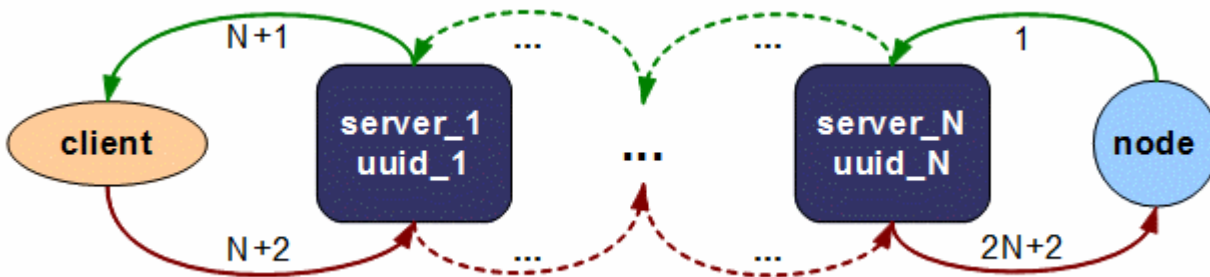
It is also important to note that this does not change anything to the class loader hierarchy within the node. In effect, there is still only one server class loader, which is associated with the server the node is *directly* connected to. This implies that the parent delegation model will not cause a class loading request to traverse the server chain multiple times.

Another implication of using client UUIDs is that it is possible to have multiple versions of the same code running within a node. Let's imagine a situation where two distinct JPPF clients, with separate UUIDs, submit the same tasks. From the node's point of view, the classes will be loaded by two distinct client class loaders, and therefore the classes from the first client will be different from those of the second client, even if they have the exact same bytecode and are downloaded from the same jar file.

The reverse situation may also happen, when two clients with the same UUID submit tasks that use different versions of the same classes. In this case, the tasks will be exposed to errors, especially at deserialization time, if the two versions are incompatible.

8.4 Built-in optimizations

JPPF provides a number of built-in optimizations and capabilities that enable to reduce the class loading overhead and avoid excessive non-heap memory consumption when the number of classes that are loaded becomes large. We will review these features in the next sections.

8.4.1 Deployment to specific grid components

In some situations, there can be a large number of classes to load before a JPPF task can be execute by a node. Even though this class loading overhead is a one-time occurrence, it can take a significant amount of time, especially if the network communication between node and server, or between server and client, is slow. This may happen, for instance, when the tasks rely on many external libraries, causing the loading of the classes within these libraries in addtiion to the classes in the application.

One way to overcome this issue is to deploy the external libraries to the JPPF server or node's classpath, to significantly reduce the time needed to load the classes in these libraries. The main drawback is that it requires to manage the deployed libraries, to ensure that they are consistently deployed across the grid, especially at such times when some of the libraries must be upgraded or removed, or new ones added. However, it is considered a good practice in production environment where few or no changes are expected during long periods of time.

8.4.2 Using a constant JPPF client UUID

In the *Development Guide*, we have seen that it is possible to set the UUID of a JPPF client to a user-defined value, using the constructor `JPPFClient(String uuid)`. This can be leveraged to force the nodes to reuse the client class loader for the specified UUID, even after the client application is terminated and has been restarted. It also implies that, if multiple clients use the same UUID, the same client class loader will also be used in the nodes. Thus, this feature limits the initial class loading overhead to the first time a job is submitted by the first client to run.

The main drawback is that, if the code of the tasks is changed on the client side, the changes will not be automatically taken into account by the nodes, and some errors may occur, due to an incompatibility between class versions in the node and in the client. If this happens, then you will have to change the client UUID or restart the nodes to force a reload of the classes by the nodes.

8.4.3 Node class loader cache

Each JPPF node maintains a cache of client class loaders. This cache has a bounded size, in order to avoid out of memory conditions caused by too many classes loaded in the JVM. This cache has an eviction policy based on the least recently created class loader. Thus, when the cache size limit is reached and a new class loader needs to be created, the oldest class loader that was created is removed from the cache, which frees up a slot for the new class loader.

As described in the Configuration Guide, the cache size is defined in the node's configuration file as follows:

```
jppf.classloader.cache.size = n
```

where n is a strictly positive integer

8.4.4 Local caching of network resources

The class loader also caches locally, either in memory or on the node's local file system, all resources found in the classpath that are not class definitions, when one of its methods `getResourceAsStream()`, `getResource()`, `getResources()` or `getMultipleResources()` is called. This avoids a potentially large network overhead the next time the same resources are requested.

The type of storage for these resources can be configured in the node's configuration file:

```
# either "file" (the default) or "memory"
jppf.resource.cache.storage = file
```

When "file" persistence is configured, the node will fall back to memory persistence if the resource cannot be saved to the file system for any reason. This could happen for instance when the file system runs out of space.

When the resources are stored on the local file system, the root of this local file cache is located at the default temp directory, such as determined by a call to System.getProperty("java.io.tmpdir"). This can be overriden using the following JPPF node configuration property:

```
jppf.resource.cache.dir = some_directory
```

In fact, the full determination of the root for the resources cache is done as follows:

- if the node configuration property "jppf.resource.cache.dir" is defined, then use its value
- otherwise, if the system property "java.io.tmpdir" is defined, then use it
- otherwise, if the system property "user.home" is defined, then use it
- otherwise, if the system property "user.dir" is defined, then use it
- otherwise, use the current directory "."

Additionally, to avoid confusion with any other applications storing temporary files, the JPPF node will store temporary resources in a directory named ".jppf" under the computed cache root. For instance, if the computed root location is "/tmp", the node will store resources under "/tmp/.jppf".

8.4.5 Batching of class loading requests

There are two distinct mechanisms that allow an efficient grouping of outgoing class laoding requests:

In the node:

Class loading requests issued by the node's processing threads are not immediately sent to the server. Instead, the node will collect requests for a very short time (by default 100 nanoseconds or more, depending on th system timer accuracy), then send them at regular intervals. While collecting requests, the node will also identify and handle duplicate requests (i.e. parallel requests from multiple threads for the same class). Thus grouped, mutliple requests (and their responses) will require much less network transport time.

The batching period can be specified with the following node configuration property:

```
# batching period for class loading requests, in nanoseconds (defaults to 100)
jppf.node.classloading.batch.period = 100
```

In the server:

A similar mechanism exists for the class loading requests forwardied by the server to a client. In this case, however, the server doesn't wait for a fixed time to send the requests. Instead it will take advantage of the time taken to send a request and receive its response. During that time, multiple nodes may be requesting the same resource, and the server will be able to send the request only once and dispatch the response to multiple nodes. The performance gains are variable but substantial: our stress tests show a class loading speedup going from 8% with 1 node, up to 30% with 50 nodes.

8.4.6 Classes cache in the JPPF server

Each JPPF server maintains an in-memory cache of classes and resources loaded via the class loading mechanism. This cache speeds up the class loading process by avoiding network lookups on the JPPF clients that hold the requested classes in their classpath. To avoid potential out of memory conditions, this cache uses soft references to store the bytecode of classes. This means that these classes may be unloaded from the cache by the garbage collector if the memory becomes scarce in the server. However, in most situations the cache still provides a significant speedup.

This cache can be disabled in the server's configuration:

```
# Specify whether the class cache is enabled. Default is 'true'
jppf.server.class.cache.enabled = false
```

8.4.7 Node customizations

As seen in the chapter *Extending and Customizing JPPF > Flow of customizations in JPPF*, most node customizations (except for the JMX logger and Initialization hooks) are loaded after the node has established a connection with the server. This enables these customizations to be loaded via the server class loader, which means they can be deployed to the server's classpath and then automatically downloaded from the server by the node.

You may also choose to deploy the customizations to the node's local classpath, in which case you will have to do it for all nodes that require this customization. In this case, the customizations will load faster but they incur the overhead of redeploying new versions to all the nodes.

8.5 Class loader delegation models

As seen previously, the JPPF class loaders follow by default the parent-first delegation model. We also saw that the base class AbstractJPPFClassloader is a subclass of URLClassLoader, which maintains a set of URLs for its classpath, each URL pointing to a jar file or class folder. A particularity of AbstractJPPFClassloader is that it overrides the addURL(URL) method to make it public instead of protected. Thus, any node customization or JPPF task will have access to this method, and will be able to dynamically extend the classpath of the JPPF class loaders.

To take advantage of this, the node provides an additional delegation model for its class loaders, which will cause them to first lookup in their URL classpath as specified with call to `addURL(URL)`, and then lookup in the remote server or client.

When this delagtion model is activated, the lookup for a class or resource from a client class loader will follow these steps:

– Lookup in the URL classpath
  • client class loader: delegate to the server class loader
  • server class loader: lookup in the URL classpath only
  • if the class is found, then end of lookup
  • otherwise, back to the client class loader, lookup in the URL classpath only
  • if the class is found, end of lookup
– Otherwise lookup in the server or client classpath
  • client class loader: delegate to the server class loader
  • server class loader: send a class loading request to the server
  • if the class is found in the server's classpath or cache, end of lookup
  • otherwise, the client class loader sends a request to the server to lookup in the client's classpath
  • if the class is found, end of lookup
  • otherwise throw a `ClassNotFoundException`

To summarize: when the URL-first delegation model is active, the node will first lookup classes and resources in the local hierarchy of URL classpaths, and then on the network via the JPPF server.

The delegation model is set JVM-wide in a node, it is not possible to specify different models for different class loader instances. There are three ways to specifiy the class loader delegation model in a node:

Statically in the node configuration:

```
# possible values: parent | url, defaults to parent
jppf.classloader.delegation = parent
```

Dynamically by API:

```
public abstract class AbstractJPPFClassLoader
   extends AbstractJPPFClassLoaderLifeCycle {

   // Determine the class loading delegation model currently in use
   public static synchronized DelegationModel getDelegationModel()

   // Specify the class loading delegation model to use
   public static synchronized void setDelegationModel(final DelegationModel model)
}
```

The delegation model is defined as the type safe enum `DelegationModel`:

```
public enum DelegationModel {
   // Standard delegation to parent first
   PARENT_FIRST,
   // Delegation to local URL classpath first
   URL_FIRST
}
```

Dynamically via JMX:

The related getter and setter are available in the interface `JPPFNodeAdminMBean`, which is also implemented by the JMX client `JMXNodeConnectionWrapper`. These allow you to dynamically and remotely change the node's delegation model:

```
public interface JPPFNodeAdminMBean extends JPPFAdminMBean {
   // Get the current class loader delegation model for the node
   DelegationModel getDelegationModel() throws Exception;

   // Set the current class loader delegation model for the node
   void setDelegationModel(DelegationModel model) throws Exception;
}
```

There is one question we are entitled to ask: *what are the benefits of using the URL-first delegation model?* The short answer is that it essentially provides a significant speedup of the class loading in the node, by providing the ability to download entire jar files and libraries and adding them dynamically to the node's class path. The next section of this chapter will detail how this, among other possibilities, can be achieved.

8.6 JPPF class loading extensions

8.6.1 Dynamically adding to the classpath

As we have seen previously, the class `AbstractJPPFClassloader`, or more accurately its direct superclass `AbstractJPPFClassLoaderLifeCycle`, exposes the `addURL(URL)` method, which is a protected method in the JDK's `URLClassLoader`. This means that it is possible to add jar files or class folders to the class path of a JPPF class loader at run time.

The main benefit of this feature is that it is possible to download entire libraries, then add them to the classpath, and thus dramatically speed up the class loading in the node. In effect, for applications that use a large number of classes, downloading a jar file will take much less time than loading classes one by one from the JPPF server or client.

Furthermore, the downloaded libraries can then be stored on the node's local file system, so they don't have to be downloaded again when the node is restarted. They can also be managed automatically (with custom code) to handle new versions of the libraries and remove old ones.

8.6.2 Downloading multiple resources at once

`AbstractJPPFClassLoader` provides an additional method to download multiple resources in a single request:

```
public URL[] getMultipleResources(final String...names)
```

This is an equivalent to the `getResource(String name)` method, except that it works with multiple resources at once. The returned array of URLs may contain null values, which means the corresponding resources were not found in the class loader's classpath. The main advantage of this method is that it performs all resources lookups in a single request, which implies a single network round-trip when looking up in the server or client's classpath.

For instance this could be used to download a set of jar files and add them dynamically to the classpath, as seen in the previous section.

8.6.3 Resources lookup on the file system

When requesting a resource via one of the `getResourceAsStream()`, `getResource()`, `getResources()` or `getMultipleResources()` methods, the JPPF class loader will lookup the specified resources in the server or client's local file system if they are not found in the class path.

This is provided as a basic convenient way to download files from a JPPF server or client, without having to use or code a specific file download faciliy (such as having an FTP server on the JPPF server of client).

However, there is a limitation to this facility: the resource path should always be relative to the server or client's current directory (determined via a `System.getProperty("user.dir")` call). In particular, using an absolute path will lead to unpredictable results.

8.6.4 Resetting the node's current task class loader

As JPPF class loader instances and their connection to the driver are separate entities, it is possible to create new client class loader instances, *for the same client UUID*, at some points in the node's life cycle. This provides the ability to use an entirely different class path for jobs submitted by the same client.

This is done by calling the Node.resetTaskClassLoader() method, which is available in two node extension points:
   • in the extended node life cycle listener, where the node is available from the jobHeaderLoaded() notifications
   • in pluggable node MBeans, where the node is provided in the MBean provider's createMBean() factory method

8.7 Related sample

Please look at the Extended Class Loading sample in the JPPF samples pack.

9 J2EE Connector

9.1 Overview of the JPPF Resource Adapter

9.1.1 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.

9.1.2 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

9.1.3 Architecture
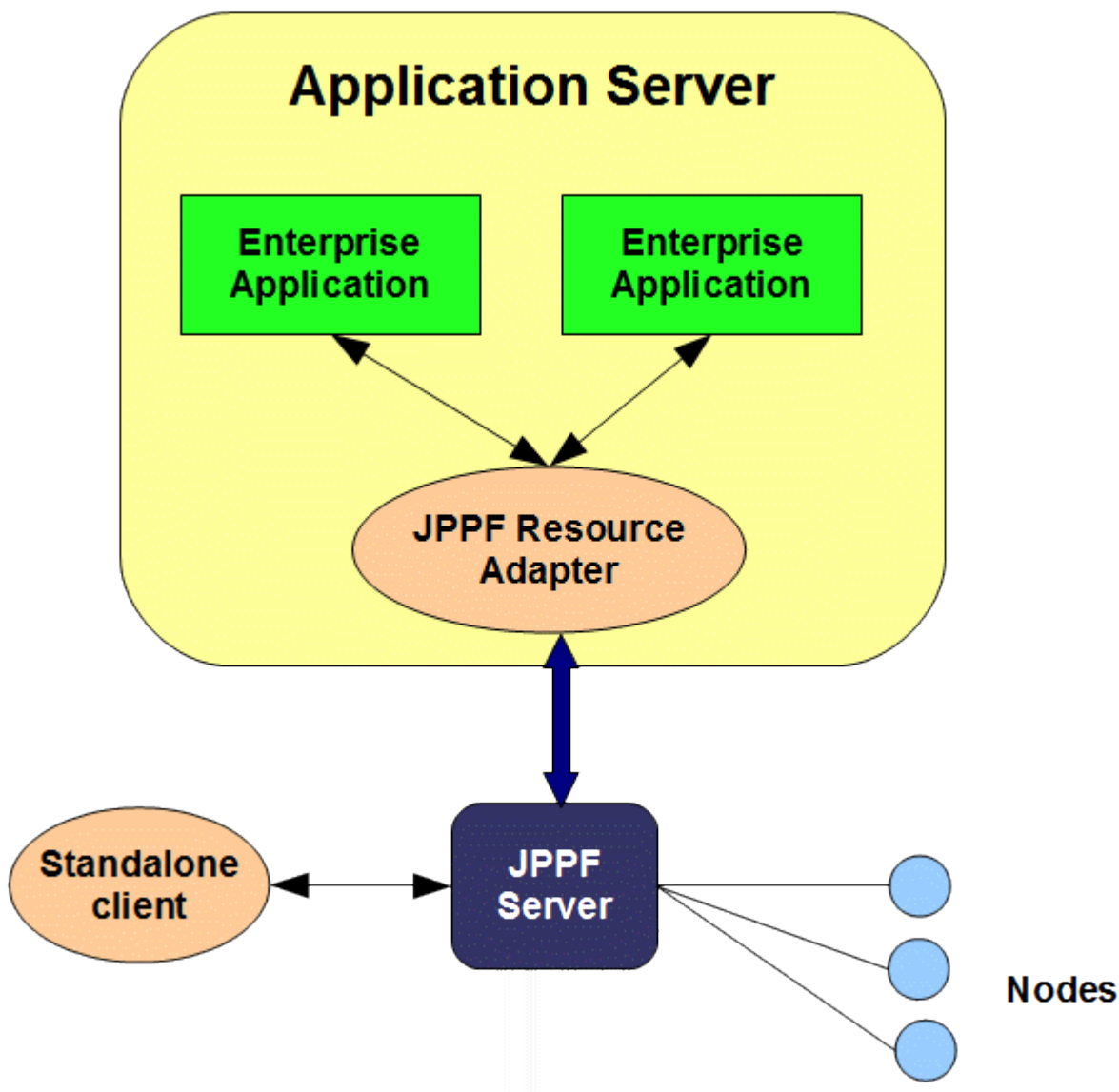
9.2 Supported Platforms

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 1.7 or later All Linux and Unix systems supporting JDK 1.7 or later
JVM	Sun JDK 1.7 and later IBM JVM 1.7 and later BEA JRockit 1.7 and later
Application Server	JBoss 4.x, 5.x, 6.x, 7.x Glassfish 2.x, 3.x IBM Websphere Application Server 8.5+ Oracle Weblogic 9.x, 10.x, 11.x Apache Geronimo 2.1.x , 3.0.x

9.3 Configuration and build

9.3.1 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"'

9.3.2 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.

9.3.3 Building the JPPF resource adapter

To build the resource adapter:

- Open a command prompt
- Go to the **JPPF-x.y-j2ee-connector** folder
- Enter "**ant build**"
- The resulting .rar and .ear files are generated in the "**build**" subfolder.

9.3.4 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:
	
	`<!-- JPPF Client Configuration -->`
	`<config-property>`
	 `<description>`Defines how the JPPF configuration is to be located.
	 This property is defined in the format "type|path", where "type" can be one of:
	 - "classpath": "path" is a path to a properties file in one of the jars of the .rar file
	  example: "classpath|resources/config/jppf.properties"
	 - "url": "path" is a url that points to a properties file
	  example: "url|file:///home/me/jppf/jppf.properties" (could be a http:// or ftp:// url as well)
	 - "file": "path" is considered a path on the file system
	  example: "file|/home/me/jppf/config/jppf.properties"
	 When no value or an invalid value is specified, "classpath|jppf.properties" is used
	 `</description>`
	 `<config-property-name>`ConfigurationSource`</config-property-name>`
	 `<config-property-type>`java.lang.String`</config-property-type>`
	 `<config-property-value>`classpath|jppf.properties`</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 for the JPPF connection factory. For example: web.xml and jboss-web.xml.

9.4 How to use the connector API

9.4.1 Obtaining and closing a resource adapter connection

The J2EE connector is accessed via the `JPPFConnection` interface. This implies that any operation performed should follow these steps:

1. obtain a connection from the resource adapter connection pool
2. perform one or more JPPF-related operation(s)
3. close the connection

The following helper code illustrates how to obtain and release connections from the resource adapter:

```java
import javax.naming.*;
import javax.resource.ResourceException;
import javax.resource.cci.ConnectionFactory;

public class JPPFHelper {
  // JNDI name of the JPPFConnectionFactory.
  public static final String JNDI_NAME = "eis/JPPFConnectionFactory";

  // Obtain a JPPF connection from the resource adapter's connection pool
  public static JPPFConnection getConnection()
    throws NamingException, ResourceException {
    // Perform a JNDI lookup of the JPPF connection factory
    InitialContext ctx = new InitialContext();
    JPPFConnectionFactory factory;
    Object objref = ctx.lookup(JNDI_NAME);
    if (objref instanceof JPPFConnectionFactory) {
      factory = (JPPFConnectionFactory) objref;
    } else {
      factory = (JPPFConnectionFactory) javax.rmi.PortableRemoteObject.narrow(
        objref, ConnectionFactory.class);
    }
    // get a JPPFConnection from the connection factory
    return (JPPFConnection) factory.getConnection();
  }

  // Release a connection
  public static void closeConnection(JPPFConnection conn) throws ResourceException {
    conn.close();
  }
}
```

Please note that the actual JNDI name for the JPPF connection factory depends on which application server is used:

– on Apache Geronimo: "**jca:/JPPF/jca-client/JCAManagedConnectionFactory/eis/JPPFConnectionFactory**"
– on JBoss: "**java:eis/JPPFConnectionFactory**"
– on Websphere: "**java:comp/env/eis/JPPFConnectionFactory**"
– on all other supported servers: "**eis/JPPFConnectionFactory**"

9.4.2 Reset of the JPPF client

The method JPPFConnection.resetClient() will trigger a reset of the underlying JPPF client. This method enables reloading the JPPF configuration without having to restart the application server. Example use:

```java
JPPFConnection connection = null;
try {
  connection = JPPFHelper.getConnection();
  connection.resetClient();
} finally {
  if (connection != null) JPPFHelper.closeConnection(connection);
}
```

As for JPPFClient.reset(), calling this method will not lose any already submitted jobs. Instead, the JCA connectior will resubmit them as soon as it is reset and server connections become available.

9.4.3 Submitting jobs

JPPFConnection provides two methods for submitting jobs:

```
public interface JPPFConnection extends Connection, JPPFAccessor {
  // Submit a job to the JPPF client
  // This method exits immediately after adding the job to the queue
  String submit(JPPFJob job) throws Exception;

  // Submit a job to the JPPF client and register the specified status listener
  // This method exits immediately after adding the job to the queue
  String submit(JPPFJob job, SubmissionStatusListener listener) throws Exception;
}
```

You will note that both methods actually perform an asynchronous job submission. They return a unique id for the the submission, which is in fact the job UUID. This id is then used to retrieve the job results and its status.

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.

```
public String submitJob() throws Exception {
  JPPFConnection connection = null;
  try {
    // get a JPPF Connection
    connection = JPPFHelper.getConnection();
    // create a JPPF job
    JPPFJob job = new JPPFJob();
    job.addTask(new DemoTask());
    // Use the connection to submit the JPPF job and obtain a submission ID
    return connection.submit(job);
  } finally {
    // close the connection
    JPPFHelper.closeConnection(connection);
  }
}
```

9.4.4 Getting the status and results of a job

Here, we check on the status of a job and process the execution results or the resulting error:

```
public void checkStatus(String submitId) throws Exception {
  JPPFConnection connection = null;
  try {
    connection = JPPFHelper.getConnection();
    // Use the connection to check the status from the submission ID
    SubmissionStatus status = connection.getSubmissionStatus(submitID);
    if (status.equals(SubmissionStatus.COMPLETE)) {
      // if successful process the results
      List<Task<?>> results = connection.getResults(submitID);
    } else if (status.equals(SubmissionStatus.FAILED)) {
      // if failed process the errors
    }
  } finally {
    JPPFHelper.closeConnection(connection);
  }
}
```

9.4.5 Cancelling a job

The J2EE allows cancelling a job by calling the method JPPFConnection.cancelJob(String submitId):

```
public void cancelJob(String submitId) throws Exception {
  JPPFConnection connection = null;
  try {
    connection = JPPFHelper.getConnection();
    // cancel the job
    connection.cancelJob(submitID);
  } finally {
    JPPFHelper.closeConnection(connection);
  }
}
```

9.4.6 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 awaitResults(String submitID), which waits until the job has completed and returns the execution results. Here is an example use:

```
// Submit a job and return the execution results
public List<JTask<?>> submitBlockingJob() throws Exception {
  List<Task<?>> results = null;
  JPPFConnection connection = null;
  try {
    connection = JPPFHelper.getConnection();
    // create a new job
    JPPFJob job = new JPPFJob();
    job.setName("test job");
    // add the tasks to the job
    for (int i=0; i<5; i++) job.add(new MyTask(i));
    // submit the job and get the submission id
    String submitID = connection.submit(job);
    // wait until the job has completed
    results = connection.awaitResults(submitID);
  } finally {
    JPPFHelper.closeConnection(connection);
  }
  // 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 time out and roll back, if the execution time is longer than the transaction timeout.

9.4.7 Using submission status events

With the J2EE connector, It is possible to subscribe to events occurring during the life cycle of a job.
This can be done via the following two methods:

```
public interface JPPFConnection extends Connection, JPPFAccessor {
  // Add a status listener to the submission with the specified id
  void addSubmissionStatusListener(
    String submissionId, SubmissionStatusListener listener);

  // Submit a job to the JPPF client and register a status listener
  String submit(JPPFJob job, SubmissionStatusListener listener) throws Exception;
}
```

Note that submit(JPPFJob, SubmissionStatusListener) submits the job *and* registers the listener in a single atomic operation. As the job submssion is asynchronous, this ensures that no event is missed between the submission of the job and the registration of the listener.

The interface `SubmissionStatusListener` is defined as follows:

```
public interface SubmissionStatusListener extends EventListener {
  // Called when the status of a job has changed
  void submissionStatusChanged(SubmissionStatusEvent event);
}
```

Each listener receives events of type `SubmissionStatusEvent`, defined as follows:

```
public class SubmissionStatusEvent extends EventObject {
  // get the status of the job
  public SubmissionStatus getStatus()

  // get the id of the job
  public String getSubmissionId()
}
```

The possible statuses are defined in the enumerated type `SubmissionStatus`:

```
public enum SubmissionStatus {
  SUBMITTED, // the job was just submitted.
  PENDING,   // the job is currently in the submission queue (on the client side)
  EXECUTING, // the job is being executed
  COMPLETE,  // the job execution is complete
  FAILED     // the job execution has failed
}
```

Here is an exemple usage of the status listeners:

```
public void submitWithListener() throws Exception {
  JPPFConnection connection = null;
  try {
    connection = JPPFHelper.getConnection();
    JPPFJob job = new JPPFJob();
    job.add(new DemoTask(duration));

    // a status listener can be added at submission time
    String id = connection.submit(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 job has 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 successful completion
            break;
          case FAILED:// process failure
            break;
          default:
            System.out.println("job [" + id + "] changed to '" + status + "'");
            break;
        }
      }
    });
    List<Task<?>> results = connection.awaitResults(id);
    // ... process the results ...
  } finally {
    JPPFHelper.closeConnection(connection);
  }
}
```

9.5 Deployment on a J2EE application server

9.5.1 Deployment on JBoss 4.x - 6.x

9.5.1.1 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")

9.5.1.2 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
<?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.

9.5.1.3 Deploying the demo application

Copy the file "JPPF_J2EE_Demo_JBoss-4.0.ear" in the <JBOSS_HOME>/server/<your_server>/deploy folder

9.5.2 Deployment on JBoss 7+

9.5.2.1 Deploying the resource adapter

- in the JBoss administration console, go to the "Deployments" view
- remove any existing JPPF deployments (.rar and .ear)
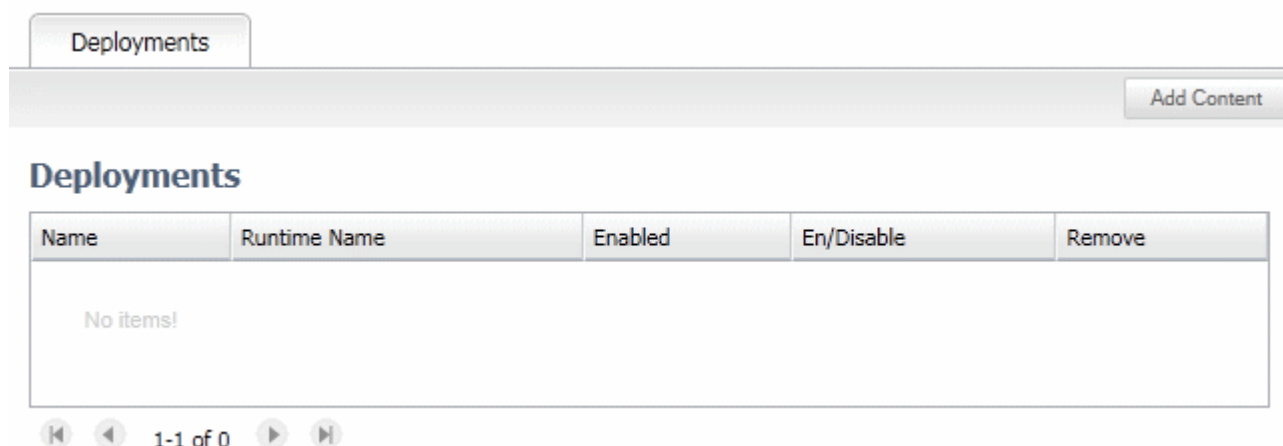
- click on "Add Content"
- browse to the file "jppf_ra_JBoss-7.rar"

Upload

Step1/2: Deployment Selection

Please choose a file that you want to deploy.

ild\jppf_ra_JBoss-7.rar Browse...

Next » Cancel

- click "next"

Upload

Step 2/2: Verify Deployment Names

Key: Cn0TGRcXtxOgUMHCA09CusCBzVw=

Name: jppf_ra_JBoss-7.rar

Runtime Name: jppf_ra_JBoss-7.rar

Finish Cancel

- accept the default names and click "Finish"
- back to the "Deployments" view, you should see the deployed resource adapter:

Deployments

Add Content

Deployments

Name	Runtime Name	Enabled	En/Disable	Remove
jppf_ra_JBoss-7.rar	jppf_ra_JBoss-7.rar	🔴	Enable	Remove

⏮ ◀ 1-1 of 1 ▶ ⏭

- click on the "Enable" button to start it:

Deployments

Add Content

Deployments

Name	Runtime Name	Enabled	En/Disable	Remove
jppf_ra_JBoss-7.rar	jppf_ra_JBoss-7.rar	🟢	Disable	Remove

⏮ ◀ 1-1 of 1 ▶ ⏭

9.5.2.2 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:

9.5.3 Deployment on SunAS / Glassfish

9.5.3.1 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"

Sun Java™ System Application Server Admin Console

Sun™ Microsystems, Inc.

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	

Common Tasks

- Application Server
  - **Applications**
    - Enterprise Applications
    - Web Applications
    - EJB Modules
    - **Connector Modules**
    - Lifecycle Modules
    - App Client Modules
  - Web Services
  - Custom MBeans
  - Resources
  - Configuration

9.5.3.2 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"

9.5.3.3 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

9.5.3.4 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

Sun Java™ System Application Server Admin Console

Sun™ Microsystems, Inc.

Application Server > Applications > Enterprise Applications

Common Tasks

- Application Server
- ▽ 📁 **Applications**
  - 📁 **Enterprise Applications**
  - 📁 Web Applications
  - 📁 EJB Modules
  - ▷ 📁 Connector Modules
  - 📁 Lifecycle Modules
  - 📁 App Client Modules
- 🌐 Web Services
- 🫘 Custom MBeans
- ▷ 🔧 Resources
- ▷ 📋 Configuration

Deploy Enterprise Application (Step 2 of 2)

Finish | Cancel

By default,the module is available as soon as it is deployed. To disable the module so that is unavailable after deployment, uncheck the Enabled checkbox. 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_J2EE_Demo_SunAS-9.0.ear
* **Application Name:**	JPPF_J2EE_Demo_SunAS-9
	Name must contain only alphanumeric, underscore, dash, or dot characters
Virtual Servers:	server
	Associates an internet domain name with a physical server
Status:	☑ Enabled
Java Web Start:	☐ Enabled
Run:	☐ Verifier
	Perform detailed verification before deploying
Precompile:	☐ JSPs
	Precompile JSPs, deploy only resulting class files
Libraries:	
Description:	
	Add a description of the component

Advanced

Generate:	☐ RMIStubs
	Generate static RMI stubs and put in client.jar

- click "Finish"
- restart the application server

9.5.4 Deployment on Websphere

9.5.4.1 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"

9.5.4.2 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

9.5.4.3 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

9.5.5 Deployment on Weblogic

9.5.5.1 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

Change Center

View changes and restarts

No pending changes exist. Click the Release Configuration button to allow others to edit the domain.

Lock & Edit

Release Configuration

Domain Structure

wls920_domain
  ⊞ Environment
  ─ Deployments
  ⊞ Services
  ─ Security Realms
  ⊞ Interoperability
  ⊞ Diagnostics

How do I...

■ Start and stop a deployed Enterprise application
■ Configure an Enterprise application
■ Create a deployment plan
■ Target an Enterprise application to a server
■ Test the modules in an Enterprise application

System Status

Health of Running Servers

	Failed (0)
	Critical (0)
	Overloaded (0)

Welcome, lcohen   Connected to: **wls920_domain**   🏠 Home | Log Out | Preferences | Help | As

Home > Summary of Deployments > jppf_ra_Weblogic-9 > Summary of Environment > **Summary of Deployments**

Install Application Assistant

Back | Next | Finish | Cancel

Locate deployment to install and prepare for deployment

Select the file path that represents the application root directory, archive file, exploded archive directory, or application module descriptor that you want to install.

Note: Only valid file paths are displayed below. If you cannot find your deployment files, upload your file(s) and/or confirm that your application contains the required deployment descriptors.

Location: 192.168.0.4 \ D: \ Workspaces \ SourceForge \ jca-client \ build

	📁 lib
◯	🔩 JPPF_J2EE_Demo_JBoss-4.0.ear
◯	🔩 JPPF_J2EE_Demo_Oracle-10.ear
◯	🔩 JPPF_J2EE_Demo_SunAS-9.0.ear
◯	🔩 JPPF_J2EE_Demo_WAS-6.1.ear
◯	🔩 JPPF_J2EE_Demo_Weblogic-9.2.ear
◯	📄 jppf_ra_JBoss-4.0.rar
◯	📄 jppf_ra_Oracle-10.rar
◯	📄 jppf_ra_SunAS-9.0.rar
◯	📄 jppf_ra_WAS-6.1.rar
◉	📄 jppf_ra_Weblogic-9.2.rar

Back | Next | Finish | Cancel

- 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.*

9.5.5.2 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

Change Center

View changes and restarts

No pending changes exist. Click the Release Configuration button to allow others to edit the domain.

Lock & Edit

Release Configuration

Domain Structure

wls920_domain
⊞ Environment
··Deployments
⊞ Services
··Security Realms
⊞ Interoperability
⊞ Diagnostics

How do I...

System Status

Welcome, lcohen   Connected to: **wls920_domain**   🏠 Home | Log Out | Preferences | Help | AskBEA

Home > **Summary of Deployments**

Install Application Assistant

Back | Next | Finish | Cancel

Locate deployment to install and prepare for deployment

Select the file path that represents the application root directory, archive file, exploded archive directory, or application module descriptor that you want to install.

Note: Only valid file paths are displayed below. If you cannot find your deployment files, upload your file(s) and/or confirm that your application contains the required deployment descriptors.

Location: 192.168.0.4 \ D: \ Workspaces \ SourceForge \ jca-client \ build

	📁 lib
○	JPPF_J2EE_Demo_JBoss-4.0.ear
○	JPPF_J2EE_Demo_Oracle-10.ear
○	JPPF_J2EE_Demo_SunAS-9.0.ear
○	JPPF_J2EE_Demo_WAS-6.1.ear
⊙	JPPF_J2EE_Demo_Weblogic-9.2.ear
○	jppf_ra_JBoss-4.0.rar
○	jppf_ra_Oracle-10.rar
○	jppf_ra_SunAS-9.0.rar
○	jppf_ra_WAS-6.1.rar
○	jppf_ra_Weblogic-9.2.rar

Back | Next | Finish | Cancel

- 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

9.5.6 Deployment on Apache Geronimo

9.5.6.1 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"

9.5.6.2 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"

9.6 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.

9.7 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-x.y-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:

```xml
<!-- ========================================================================== -->
<!-- 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="jboss7"      value="JBoss-7"/>
<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.

10 Configuration properties reference

10.1 Server properties

Property name	Default Value	Comments
jppf.server.port	11111	JPPF server port
jppf.management.enabled	true	enable server management
jppf.management.host	computed	management server host
jppf.management.port	11198	management remote connector 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	named peer server host name or address
jppf.peer.<name>.server.port	11111	named peer server port
jppf.peer.dicovery.enabled	false	enable peer discovery
jppf.load.balancing.algorithm	proportional	load balancing algorithm name
jppf.load.balancing.profile	jppf	load balancing parameters profile name
jppf.load.balancing.profile.<profile>.<parameter>	null	parameter for the named parameters profile
jppf.jvm.options	null	JVM options for the server process
jppf.transition.thread.pool.size	available processors	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	available processors	number of threads allocated to the reaper
jppf.nio.connection.check	true	enable network connection checks on write operations

jppf.discovery.broadcast.include.ipv4	null	broadcast to the specified IPv4 addresses (inclusive filter)
jppf.discovery.broadcast.exclude.ipv4	null	don't broadcast to these IPv4 addresses (exclude filter)
jppf.discovery.broadcast.include.ipv6	null	broadcast to the specified IPv6 addresses (inclusive filter)
jppf.discovery.broadcast.exclude.ipv6	null	don't broadcast to these IPv6 addresses (exclude filter)
jppf.ssl.server.port	11443	port number for secure connections
jppf.peer.ssl.enabled	false	toggle secure connections to remote peer servers
jppf.management.ssl.enabled	false	enable JMX via secure connections
jppf.management.ssl.port	11193	secure JMX server port

10.2 Node properties

Property name	Default Value	Comments
jppf.server.host	localhost	JPPF server host address
jppf.server.port	11111	JPPF server 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.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.include.ipv4	null	IPv4 inclusion patterns for server discovery
jppf.discovery.exclude.ipv4	null	IPv4 exclusion patterns for server discovery
jppf.discovery.include.ipv6	null	IPv6 inclusion patterns for server discovery
jppf.discovery.exclude.ipv6	null	IPv6 exclusion patterns for server discovery
jppf.jvm.options	null	JVM options for the node process
jppf.processing.threads	available processors	number of threads used for tasks execution
jppf.policy.file	null	path to the security policy file, in the node's classpath or 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)	frequency of checks for keyboard and mouse inactivity, in ms
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.node.offline	false	whether the node runs in offline mode
jppf.ssl.enabled	false	toggle secure connections

10.3 Node screen saver properties

Property name	Default Value	Comments
General properties		
jppf.screensaver.enabled	false	enable/disable the screen saver
jppf.screensaver.class	null	class name of an implementation of JPPFScreenSaver
jppf.screensaver.node.listener	null	class name of an implementation of NodeIntegration
jppf.screensaver.title	JPPF screensaver	title of the JFrame used in windowed mode
jppf.screensaver.icon	o/j/n/jppf-icon.gif *	path to the image for the frame's icon (windowed mode)
jppf.screensaver.fullscreen	false	display the screen saver in full screen mode
jppf.screensaver.width	1000	width in pixels (windowed mode)
jppf.screensaver.height	800	height in pixels (windowed mode)
jppf.screensaver.mouse.motion .close	true	close on mouse motion (full screen mode)
Built-in screen saver (*JPPFScreenSaverImpl*) properties		
jppf.screensaver.handle.collisions	true	handle collisions between moving logos
jppf.screensaver.logos	10	number of moving logos
jppf.screensaver.speed	100	speed of moving logos from 1 to 100
jppf.screensaver.logo.path	o/j/n/jppf_group_small.gif *	path(s) to the moving logo image(s)
jppf.screensaver.centerimage	o/j/n/jppf@home.gif *	path to the larger image at the center of the screen
jppf.screensaver.status.panel .alignment	center	horizontal alignment of the status panel

* o/j/n means org/jppf/node, a package/folder in jppf-common-node.jar

10.4 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	named driver's address or host name
<driver_name>.jppf.server.port	11111	named driver's port
<driver_name>.jppf.management.host	localhost	named server's management 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>.jppf.priority	0	named server priority
<driver_name>.jppf.pool.size	1	named server connection pool size
jppf.local.execution.enabled	true	enable remote execution
jppf.local.execution.enabled	false	enable local execution
jppf.local.execution.threads	available processors	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: UDP multicast group
jppf.discovery.port	11111	server discovery: UDP multicast port
jppf.discovery.include.ipv4	null	IPv4 inclusion patterns for server discovery
jppf.discovery.exclude.ipv4	null	IPv4 exclusion patterns for server discovery
jppf.discovery.include.ipv6	null	IPv6 inclusion patterns for server discovery
jppf.discovery.exclude.ipv6	null	IPv6 exclusion patterns for server discovery
jppf.socket.max-idle	-1	number of seconds a socket connection can remain idle before being closed
jppf.ui.splash	true	enable display of splash screen at startup
jppf.local.execution.accumulation.size	all tasks	how many completed tasks before notifying
jppf.local.execution.accumulation.time	job completion time	how long before notifying
jppf.local.execution.accumulation.unit	m = milliseconds	time unit to use
jppf.ssl.enabled	false	toggle secure connections

10.5 Common configuration properties

Property name	Default Value	Comments
jppf.reconnect.initial.delay	1	delay in seconds before the first reconnection attempt
jppf.reconnect.max.time	60	delay in seconds after which reconnection attemps stop.
jppf.reconnect.interval	1	frequency in seconds of reconnection attempts
jppf.object.serialization.class	DefaultJavaSerialization	serialization scheme, a class implementing JPPFSerialization
jppf.data.transform.class	null	optional network data transformation
jppf.socket.buffer.size	32768	receive/send buffer size for socket connections
jppf.socket.tcp_nodelay	true	disable Nagle's algorithm
jppf.socket.keepalive	false	enable / disable keepalive
jppf.temp.buffer.size	32768	size of temporary buffers used in I/O transfers
jppf.ssl.configuration.file	null	SSL configuration in the file system or classpath
jppf.ssl.configuration.source	null	SSL configuration as an arbitrary source

10.6 SSL properties

Property name	Default Value	Comments
jppf.ssl.context.protocol	SSL	SSLContext protocol
jppf.ssl.protocols	null	a list of space-separated enabled protocols
jppf.ssl.cipher.suites	null	enabled cipher suites as space-separated values
jppf.ssl.client.auth	none	SSL client authentication level
jppf.ssl.keystore.file	null	path to the key store in the file system or classpath
jppf.ssl.keytstore.source	null	key store location as an arbitrary source
jppf.ssl.keystore.password	null	plain text key store password
jppf.ssl.keyststore.password.source	null	key store password as an arbitrary source
jppf.ssl.truststore.file	null	path to the trust store in the file system or classpath
jppf.ssl.truststore.source	null	trust store location as an arbitrary source
jppf.ssl.truststore.password	null	plain text trust store password
jppf.ssl.truststore.password.source	null	trust store password as an arbitrary source
jppf.ssl.client.distinct.truststore	false	use a separate trust store for client certificates (server only)
jppf.ssl.client.truststore.file	null	path to the client trust store in the file system or classpath
jppf.ssl.client.truststore.source	null	client trust store location as an arbitrary source
jppf.ssl.client.truststore.password	null	plain text client trust store password
jppf.ssl.client.truststore.password.source	null	client trust store password as an arbitrary source

11 Execution policy reference

11.1 Execution Policy Elements

11.1.1 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>
```

11.1.2 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>
```

11.1.3 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>
```

11.1.4 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>
```

11.1.5 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>
```

11.1.6 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>
```

11.1.7 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>
```

11.1.8 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>
```

11.1.9 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>
```

11.1.10 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>
```

11.1.11 BetweenIE

Performs a test of type "property_value in [a, b[" (lower bound included, upper bound 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>
```

11.1.12 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>
```

11.1.13 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>
```

11.1.14 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>
```

11.1.15 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", "fr", "it");
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>fr</Value>
  <Value>it</Value>
</OneOf>
```

11.1.16 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>
```

11.1.17 ScriptedPolicy

Executes a script which returns a boolean value.

Class name: **org.jppf.node.policy.ScriptedPolicy**
Constructors:
```
ScriptedPolicy(String language, String script)
ScriptedPolicy(String language, Reader scriptReader)
ScriptedPolicy(String language, File scriptFile)
```
Usage:
```
policy = new ScriptedPolicy("javascript", "true");
policy = new ScriptedPolicy("javascript", new StringReader(myScript));
policy = new ScriptedPolicy("javascript", new File("myScript.js"));
```

XML Element: **<Script>**
Attribute: language
Usage:
```
<Script language="javascript">true</Script>

<Script language="javascript"><![CDATA[
function myFunction() {
  return true;
}
myFunction();
]]></Script>
```

11.1.18 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>
```

11.2 Execution policy properties

11.2.1 Related APIs

All 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 or driver connection. 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:

8. in JPPFSystemInformation.getUuid() : JPPF uuid and version properties
9. in JPPFSystemInformation.getJppf() : JPPF configuration properties
10. in JPPFSystemInformation.getSystem() : system properties
11. in JPPFSystemInformation.getEnv() : environment variables
12. in JPPFSystemInformation.getNetwork() : IPV4 and IPV6 addresses assigned to the node or driver
13. in JPPFSystemInformation.getRuntime() : runtime properties
14. in JPPFSystemInformation.getStorage() : storage space properties

11.2.2 JPPF uuid and version properties

The following properties are provided:

`jppf.uuid` : the uuid of the node or driver
`jppf.version.number` : the current JPPF version number
`jppf.build.number` : the current build number
`jppf.build.date` : the build date, including the time zone, in the format "yyyy-MM-dd hh:mm z"

Related APIs:

JPPFSystemInformation.getUuid()
VersionUtils.getVersion()

11.2.3 JPPF configuration properties

The JPPF properties are all the properties defined in the node's or driver's JPPF configuration file, depending on where the execution policy applies.

Additionally, there is one special property "**jppf.channel.local**", which is set internally by JPPF and which determines whether the job executor is a local node (i.e. node local to the driver's JVM) when used in a server SLA, or a local executor in the client when used in a client SLA. When used in a client SLA, this allows toggling local vs. remote execution on a per-job basis, as in the following example:

```
JPPFJob job = ...;
// allow job execution only in the client-local executor
ExecutionPolicy localExecutionPolicy = new Equal("jppf.channel.local", true);
job.getClientSLA().setExecutionPolicy(localExecutionPolicy);
```

Related APIs:

JPPFSystemInformation.getJppf()
JPPFConfiguration.getProperties()

11.2.4 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()

11.2.5 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()

11.2.6 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*.

11.2.7 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

11.2.8 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 information. 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
```

11.3 Execution policy XML schema

```xml
<?xml version="1.0" encoding="UTF-8"?>

<!--
  JPPF.
  Copyright (C) 2005-2014 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"/>
      <element name="Script" type="jppf:ScriptedRuleType"/>
    </choice>
  </group>

  <!-- unary predicates : NOT -->
  <complexType name="OneRuleType">
    <sequence>
      <group ref="jppf:Rule"/>
    </sequence>
  </complexType>

  <!-- binary predicates : AND, OR, XOR -->
  <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>
```

```xml
  <simpleType name="TrueFalse">
    <restriction base="string">
      <enumeration value="true"/>
      <enumeration value="false"/>
    </restriction>
  </simpleType>

  <!-- custom execution policy -->
  <complexType name="CustomRuleType">
    <sequence>
      <element name="Arg" type="string" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="class" type="string"/>
  </complexType>

  <!-- scripted execution policy -->
  <complexType name="SciptedRuleType">
    <simpleContent>
      <extension base="string">
        <attribute name="language" type="string"/>
      </extension>
    </simpleContent>
  </complexType>

</schema>
```

12 Deployment and run modes

12.1 JPPF Driver

First you will need to obtain the driver package from the <u>JPPF download page</u> : **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**".

12.1.1 JPPF Driver as a Windows Service

A JPPF driver can be run as Windows Service using the Java Service Wrapper available at <u>Tanuki Software</u>.
The JPPF-x.y.z-driver.zip distribution and above are prepared for this installation.

To install:

- <u>download</u> 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

12.1.2 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 <u>Tanuki Software</u>.
The JPPF-x.y.z-driver.zip distribution and above are prepared for this installation.

To install:

- <u>download</u> 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**

12.2 JPPF Node

First you will need to obtain the node package from the <u>JPPF download page</u> : **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**".

12.2.1 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:

- <u>download</u> 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

12.2.2 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**

12.2.3 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

12.3 Running JPPF on Amazon's EC2

12.3.1 Server discovery

EC2 does not allow multicast network communication, so JPPF nodes must know which server to use ahead of time instead of using the auto-discovery feature. So the server property file must set:

```
jppf.discovery.enabled = false
jppf.peer.discovery.enabled = false
```

And the node property file must set:

```
jppf.discovery.enabled = false
jppf.server.host = IP_or_DNS_hostname
```

Similarly the client must set:

```
jppf.discovery.enabled = false
jppf.drivers = driverA
driverA.jppf.server.host = IP_or_DNS_hostname
driverA.jppf.server.port = 11111
```

Amazon charges for network access to a public IP, so you'll want the node to communicate with the internal 10.x.x.x address and not a public IP. More on this detail below...

12.3.2 Firewall configuration

EC2 puts all nodes into "security groups" that define allowed network access. Make sure to start JPPF servers with a special security group that allows access to the standard port 11111 and if you use the management tools remotely, 11198. You may also want to limit these to internal IPs 10.0.0.0/8 if your clients, servers and nodes are all within EC2.

12.3.3 Instance type

EC2 nodes vary the number of available cores, so you may want a different node property file for each instance type you start, with an appropriate number of threads. For instance, on a c1.xlarge instance with 8 cores, you might want to have one additional thread so the CPU would be busy if any one thread was waiting on I/O:

```
processing.threads = 9
```

If your tasks require more I/O, you may need to experiment to find the best completion rate.

12.3.4 IP Addresses

All EC2 instances will have both a public IP address (chosen randomly or your selected elastic IP), and a private internal IP 10.x.x.x. You are charged for traffic between availability zones regardless of address, and even within the same zone if you use the external IP. So you'll want to try to have the systems connect using the 10.x.x.x addresses.

Unfortunately, this complicates things a bit. Ideally you probably want to set up a pre-configured node image (AMI) and launch instances from that image as needed for your JPPF tasks. But you may not know the internal IP of the driver at the time. And you don't want to spend time creating a new AMI each time you launch a new task with a new driver.

One solution is to use a static elastic IP that you will always associate with the JPPF driver and eat the cost of EC2 traffic. It isn't that much really...

Or you can use DNS to publish your 10.x.x.x IP address for the driver before launching nodes, and configure the node AMI to use a fixed DNS hostname.

Or you can do a little programming with the EC2 API to pass the information around. This is the recommended approach. To this effect, JPPF provides a configuration hook, which will allow a node to read its configuration from a source other than a static and local configuration file. The node configuration plugin can read a properties file from S3 instead of a file already on the node. A matching startup task on the driver instance would publish an appropriate properties file to S3. There are lots of other approaches that will give you the same results – just have the server publish its location to some known location and have the node read this and dynamically create its properties instead of having a fixed file.

12.4 Offline nodes

JPPF 4.0 introduced a new "offline" mode for the nodes: in this mode, the nodes will disconnect from the server before executing the tasks, then reconnect to send the results and get a new set of tasks. The tasks are thus executed offline. As a consequence, the distributed class loader connection is disabled, and so is the JMX-based remote management of the node.

In this mode, the scalability of a single-server JPPF grid is greatly increased, since it becomes possible to have many more nodes than there are available TCP ports on the server. This is also particularly adapted to a volunteer computing type of grid, especially when combined with the "Idle Host" mode.

12.4.1 Class loading considerations

Since the dynamic class loader is disabled for offline nodes, we need another way to ensure that the tasks and supporting classes are known to the node. There are two solutions for this, which can be used together:

- statically: the classes can be deployed along with each node and added to its classpath
- dynamically: the supporting libraries can be sent along with the jobs, using the job's SLA classpath attribute. The nodes have a specific mechanism to add these libraries to the classpath before deserializing and executing the tasks.

Additionally, the node will also need the JPPF libraries used by the JPPF server in its classpath: *jppf-server.jar* and *jppf-common.jar*.

12.4.2 Avoiding stuck jobs

Since offline nodes work disconnected most of the time, the server has no way to know the status of a node, nor detect whether it crashed or is unable to reconnect. When this happens, the standard JPPF recovery mechanism, which resubmits the tasks sent to the node when a disconnection is detected, cannot be applied. In turn, this will cause the entire job to be stuck in the server queue, never completing.

To avoid this risk, the job SLA allows you to specify an expiration for all subsets of a job sent to any node. This expiration, specified as either a fixed date or a maximum duration for the job dispatch (i.e. subset of the job sent to a node), will cause the server to consider the job dispatch to have failed, and resubmit or simply cancel the tasks it contains, depending on the maximum number of allowed expirations specified in the SLA.

12.4.3 Example: configuring an offline node and submitting a job

To configure an offline node, set the following properties in its configuration file:

```
# enable offline mode
jppf.node.offline = true
# add the JPPF server libraries to the classpath
jppf.jvm.options = -server -Xmx512m -cp lib/jppf-server.jar -cp lib/jppf-common.jar
```

Note that we specified the additional libraries as 2 distincts "-cp" statements, so that the path specifications do not depend on a platform-specific syntax. For instance on Linux we could just write instead:

```
-cp lib/jppf-server.jar:lib/jppf-common.jar
```

To submit a job along with a supporting library:

```
JPPFJob myJob = new JPPFJob();
ClassPath classpath = myJob.getSLA().getClassPath();
// wrap a jar file into a FileLocation object
Location jarLocation = new FileLocation("libs/MyLib.jar");
// copy the jar file into memory
Location location = new MemoryLocation(jarLocation.toByteArray());
// add it as classpath element
classpath.add("myLib", location);
// tell the node to reset the tasks classloader with this new class path
classpath.setForceClassLoaderReset(true);
// set the job dispatches to expire if they execute for more than 5 seconds
myJob.getSLA().setDispatchExpirationSchedule(new JPPFSchedule(5000L));
// dispatched tasks will be resubmitted at most 2 times before they are cancelled
myJob.getSLA().setMaxDispatchExpirations(2);
// submit the job
JPPFClient client = ...;
List<JPPFTask> results = client.submit(myJob);
```

12.5 Runtime dependencies

12.5.1 Node and Common dependencies

These libraries are those used by the JPPF nodes as well as by all other JPPF components. For greater clarity, they are not shown in the next sections, however they should always be added to the JPPF components' classpath.

Library name	Jar Files	Comments
SLF4J	slf4j-api-1.6.1.jar slf4j-log4j12-1.6.1.jar	Logging wrapper API with Log4j bindings
Log4j	log4j-1.2.15.jar	Logging implementation
JMXMP	jmxremote_optional.jar	JMX remote connector
JPPF node	jppf-common-node.jar	Node bootstraping API and common JPPF APIs

12.5.2 Driver dependencies

Library name	Jar Files	Comments
JPPF common	jppf-common.jar	Common JPPF APIs not needed to bootstrap a node
JPPF server	jppf-server.jar	Driver-specific code

12.5.3 Client dependencies

Library name	Jar Files	Comments
JPPF common	jppf-common.jar	Common JPPF APIs not needed to bootstrap a node
JPPF client	jppf-client.jar	Client-specific code

12.5.4 Administration console dependencies

Library name	Jar Files	Comments
JPPF common	jppf-common.jar	Common JPPF APIs not needed to bootstrap a node
JPPF client	jppf-client.jar	Client-specific code
JPPF admin	jppf-admin.jar	Administration console-specific code
Rhino	js.jar	Mozilla's JavaScript APIs and engine
Groovy	groovy-all-1.6.5.jar	Groovy scripting engine and APIs
MigLayout	miglayout-3.7-swing.jar	Swing layout library
JFreeChart	jfreechart-1.0.12.jar jcommon-1.0.15.jar	Charting components for Swing GUIs
JGoodies Looks	looks-2.2.2.jar	Swing look and feel
JUNG	jung-algorithms-2.0.1.jar jung-api-2.0.1.jar jung-graph-impl-2.0.1.jar jung-visualization-2.0.1.jar collections-generic-4.01.jar colt-1.2.0.jar concurrent-1.3.4.jar	Graph library

13 API changes in JPPF 4.0

13.1 Introduction

This chapter lists the public documented classes and methods that have changes from the latest JPPF 3.x version to JPPF 4.0. Some methods have been deprecated and should be working in existing code, to the best of our knowledge. It is strongly recommended that any new code written against the 4.0 API strictly avoid the use of deprecated APIs.

13.2 The new Task API

13.2.1 New design

In JPPF 3.x, JPPFTask was, for all practical purposes, the base class for all tasks in JPPF. Version 4.0 introduces a more generic API, based on the Task<T> interface and its concrete implementation AbstractTask<T>, with JPPFTask redefined as `public class JPPFTask extends AbstractTask<Object> {}`.

Most of of the API changes listed here relate to this new design, which has implications well beyond the scope of the tasks implementation.

As of JPPF 4.0, the base interface for concrete task implementations is offically Task<T>, with JPPF providing a generic abstract implementation AbstractTask<T>, which implements all methods of the interface, except the `run()` method inherited from the `Runnable` interface. All JPPF 4.0 users are strongly encouraged to use AbstractTask<T>, for consistency with the framework.

13.2.2 New and changed methods

In Task<T>, the methods `getException()` and `setException(Exception)` were deprecated and are now replaced with the more generic `getThrowable()` and `setThrowable(Throwable)` respectively.

Since these new methods may be inconsistent with `catch(Some_Exception_Class) {}` clauses in existing code, a utility class ExceptionUtils is provided, with helper methods to convert a generic `Throwable` into a more specfic exception type.

A new method `fireNotification(Object userObject, boolean sendViaJmx)` was added, which allows the tasks to send notifications to local listeners ot remote listeners via JMX.

13.3 Package org.jppf.client

13.3.1 Class JPPFClient

`submit(JPPFJob)`, which returns a List<JPPFTask>, is deprecated and replaced with `submitJob(JPPFJob)`, which returns a `List<Task<?>>`.

`reset()` and `reset(TypedProperties)` methods have been added.

13.3.2 Class AbstractJPPFClient

As for `JPPFClient`, the abstract definition of `submit(JPPFJob)` is deprecated and replaced with the method `submitJob(JPPFJob)`,

13.3.3 Class JPPFJob

`addTask(Object, Object...)` and `addTask(String, Object, Object...)`, which return a `JPPFTask`, are deprecated and replaced with `add(Object, Object...)` and `add(String, Object, Object...)` respectively, which return a `Task<?>`.

13.3.4 Class JPPFResultCollector

`getResults()`, `waitForResults()`, and `waitForResults(long)`, which return a `List<JPPFTask>`, have been deprecated and replaced with `getAllResults()`, `awaitResults()` and `awaitResults(long)` respectively, which return a `List<Task<?>>`.

13.3.5 Class JobResults

`getAll() : Collection<JPPFTask>` is replaced with `getAllResults() : Collection<Task<?>>`
`getResult(int) : JPPFTask` is deprecated and replaced with `getResultTask(int) : Task<?>`
`putResults(List<JPPFTask>)` is deprecated and replaced with `addResults(List<Task<?>>)`

13.4 Package org.jppf.client.event

13.4.1 Class JobEvent

`getTasks() : List<JPPFTask>` is deprecated and replaced with `getJobTasks() : List<Task<?>>`.

13.4.2 Class TaskResultEvent

`getTaskList() : List<JPPFTask>` is deprecated and replaced with `getTasks() : List<Task<?>>`.

13.5 Package org.jppf.client.persistence

13.5.1 Class JobPersistence

`storeJob(K, JPPFJob, List<Task<?>>)` was changed into `storeJob(K, JPPFJob, List<Task<?>>)`.

13.5.2 Class DefaultFilePersistenceManager

`storeJob(K, JPPFJob, List<Task<?>>)` was changed into `storeJob(K, JPPFJob, List<Task<?>>)`.

13.6 Package org.jppf.client.taskwrapper

13.6.1 JPPFTaskCallback

The class `JPPFTaskCallback` has been genericized into `JPPFTaskCallback<T>`.
`getTask() : JPPFTask` was changed into `getTask() : Task<T>`.

13.7 Package org.jppf.client.utils

13.7.1 Class ClientWithFailover

`submit(JPPFJob): List<JPPFTask>` is deprecated and replaced with `submitJob(JPPFJob): List<Task<?>>`.

13.8 Package org.jppf.server.protocol

13.8.1 Class CommandLineTask genericized

`CommandLineTask` is now defined as `CommandLineTask<T> extends AbstractTask<T>`.

13.9 Package org.jppf.node.event

The classes `TaskExecutionListener` and `TaskExecutionEvent` moved from package `org.jppf.server.node`.

13.9.1 Class TasKExecutionListener

The method `taskNotification(TaskExecutionEvent)` has been added.

13.9.2 TasKExecutionEvent

The method `getTask() : JPPFTask` was changed into `getTask() : Task<?>`.

13.10 Package org.jppf.jca.cci (J2E connector)

13.10.1 Class JPPFConnection

`getSubmissionResults(String): List<JPPFTask>` and `waitForResults(String): List<JPPFTask>` have been deprecated and replaced with `getResults(String): List<Task<?>>` and `awaitResults(String): List<Task<?>>` respectively

The method `resetClient()` was added.

13.10.1 Class JPPFConnection

`getSubmissionResults(String): List<JPPFTask>` and `waitForResults(String): List<JPPFTask>` have been deprecated and replaced with `getResults(String): List<Task<?>>` and `awaitResults(String): List<Task<?>>` respectively

The method `resetClient()` was added.