



The open source grid computing solution

User Guide

Table of Contents

1 Introduction	3	7.3 Drivers and clients	197
1.1 Intended audience	3	7.4 Drivers	205
1.2 Prerequisites	3	7.5 Nodes	217
1.3 Where to download	3	7.6 Clients	235
1.4 Installation	3	7.7 Administration console	238
1.5 Running the standalone modules	3	7.8 Flow of customizations in JPPF	243
2 JPPF Overview	4	8 Class Loading In JPPF	244
2.1 Architecture and topology	4	8.1 How it works	244
2.2 Work distribution	5	8.2 Class loader hierarchy in JPPF nodes	245
2.3 Jobs and tasks granularity	6	8.3 Relationship between UUIDs and class loaders	246
2.4 Networking considerations	7	8.4 Built-in optimizations	247
2.5 Sources of parallelism	9	8.5 Class loader delegation models	249
3 Tutorial : A first taste of JPPF	10	8.6 JPPF class loading extensions	250
3.1 Required software	10	8.7 Related sample	251
3.2 Overview	10	9 Load Balancing	252
3.3 Writing a JPPF task	11	9.1 What does it do?	252
3.4 Creating and executing a job	11	9.2 Load balancing API	254
3.5 Running the application	14	9.3 Built-in algorithms	261
3.6 Dynamic deployment	15	9.4 Load-balancer state persistence	265
3.7 Job Management	15	10 Database services	270
3.8 Conclusion	18	10.1 Configuring JDBC data sources	270
4 Development Guide	19	10.2 The JPPFDatasourceFactory API	273
4.1 Task objects	19	10.3 Class loading and classpath considerations	275
4.2 Dealing with jobs	32	11 J2EE Connector	276
4.3 Jobs runtime behavior, recovery and failover	36	11.1 Overview of the JPPF Resource Adapter	276
4.4 Sharing data among tasks : the DataProvider API	39	11.2 Supported Platforms	277
4.5 Job Service Level Agreement	41	11.3 Configuration and build	277
4.6 Job Metadata	53	11.4 How to use the connector API	279
4.7 Execution policies	54	11.5 Deployment on a J2EE application server	284
4.8 The JPPFClient API	61	11.6 Packaging your enterprise application	310
4.9 Connection pools	65	11.7 Creating an application server port	310
4.10 Notifications of client job queue events	72	12 Configuration properties reference	312
4.11 Submitting multiple jobs concurrently	72	12.1 Driver properties	312
4.12 Jobs persistence in the driver	78	12.2 Node properties	313
4.13 JPPF Executor Services	86	12.3 Node screensaver properties	315
4.14 Grid topology monitoring	90	12.4 Client properties	315
4.15 Job monitoring API	96	12.5 Desktop console properties	316
4.16 The JPPF statistics API	101	12.6 Web console properties	317
4.17 The Location API	103	12.7 7 Desktop and Web consoles properties	317
4.18 Job selectors	105	12.8 Common properties	317
4.19 Job dependencies and job graphs	108	12.9 .Net properties	318
5 Configuration guide	112	12.10 SSL/TLS properties	318
5.1 Configuration file specification and lookup	112	12.11 Memory usage optimization properties	318
5.2 Includes, substitutions and scripted values in the configuration	113	13 Execution policy reference	319
5.3 Reminder: JPPF topology	116	13.1 Execution Policy Elements	319
5.4 Configuring a JPPF server	117	13.2 Execution policy properties	331
5.5 Node configuration	125	14 JPPF Deployment	335
5.6 Client and administration console configuration	130	14.1 Drivers and nodes as services	335
5.7 Common configuration properties	138	14.2 Running JPPF on Amazon's EC2, Rackspace, or other Cloud Services	336
5.8 Putting it all together	139	14.3 Nodes in "Idle Host" mode	338
5.9 Configuring SSL/TLS communications	141	14.4 Offline nodes	339
5.10 The JPPF configuration API	146	14.5 Node provisioning	340
6 Management and monitoring	150	14.6 Runtime dependencies	345
6.1 Node management	150	14.7 Web administration console	347
6.2 Server management	159	14.8 Embedded driver and node	348
6.3 Nodes management and monitoring via the driver	170	14.9 JPPF in Docker containers	350
6.4 JVM health monitoring	174	15 Changes from previous versions	351
7 Extending and Customizing JPPF	179	15.1 Changes in JPPF 6.0	351
7.1 Global extensions	179	15.2 Changes in JPPF 5.0	353
7.2 Drivers and nodes	190	15.3 API changes in JPPF 4.0	355

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 8** or later, with the environment variable `JAVA_HOME` pointing to your Java installation root folder
- **Apache Ant, version 1.9.1** 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*

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.

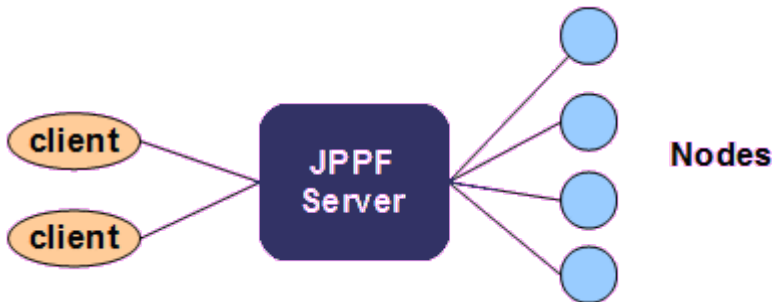
2 JPPF Overview

2.1 Architecture and topology

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

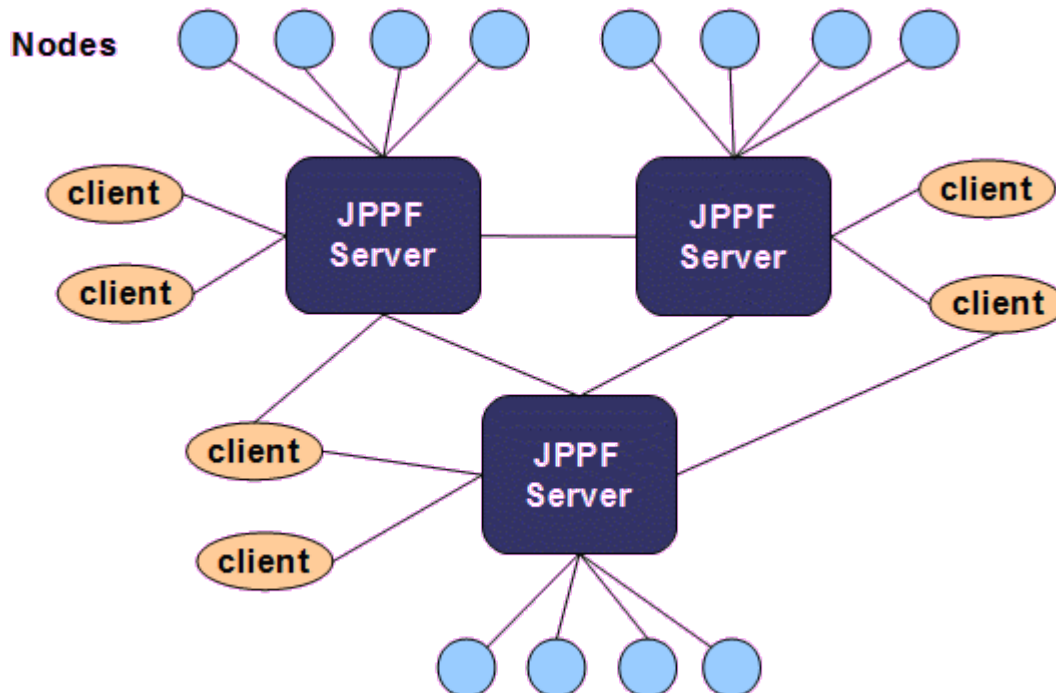
- **clients** are entry points to the grid and enable developers to submit work via the client APIs
- **servers**, also called **drivers**, are the components that receive work from the clients and dispatch it to the nodes
- **nodes** perform the actual work execution

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.

2.2 Work distribution

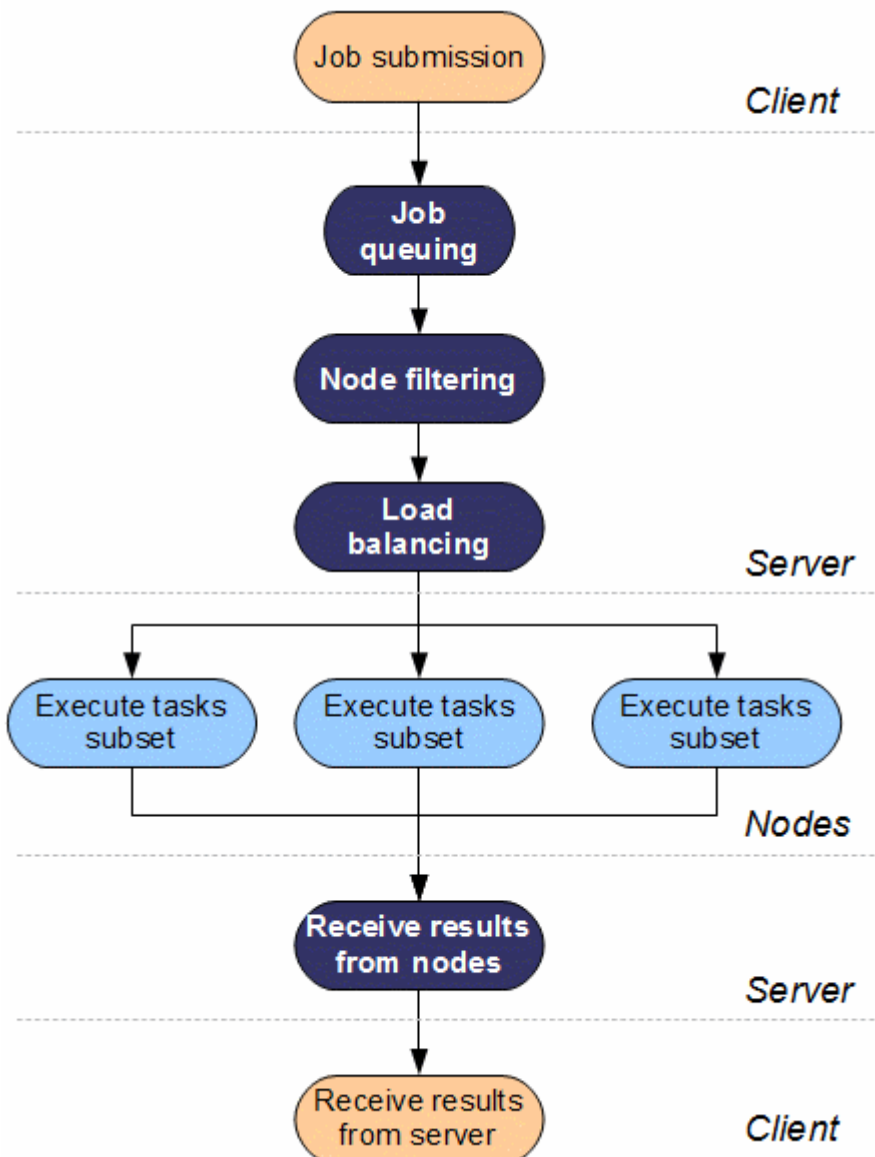
To understand how the work is distributed in a JPPF grid, and what role is played by each component, we will start by defining the two units of work that JPPF handles.

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.

2.3 Jobs and tasks granularity

The granularity of the jobs and tasks, that is, the way you divide the workload into small independent units of work, has a significant impact on performance. By design, JPPF is particularly well adapted for workloads that can be divided into many small tasks independent from each other, also known as the class of [embarrassingly parallel problems](#).

There are, however, limits to how much performance can be gained by dividing a work load. In particular, if the tasks are too small, the overhead of executing them on a grid may largely outweigh the benefits of parallelization, resulting in overall performance loss. On the other hand, if the tasks are too coarse, the execution time may be equivalent to what you'd get with a sequential execution. The granularity of the tasks must therefore be carefully considered.

To illustrate this notion, let's take an example: the multiplication of two square matrices. Let's say we have 2 square matrices A and B of size n. We define the matrix C as the result of their multiplication:

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}, B = \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \dots & \dots & \dots \\ b_{n1} & \dots & b_{nn} \end{pmatrix}$$
$$C = A \times B = \begin{pmatrix} c_{11} & \dots & c_{1n} \\ \dots & \dots & \dots \\ c_{n1} & \dots & c_{nn} \end{pmatrix}$$
$$\text{where } c_{ij} = \sum_{k=1}^n a_{ik} b_{kj} = \begin{pmatrix} a_{i1} & \dots & a_{in} \end{pmatrix} \times \begin{pmatrix} b_{1j} \\ \dots \\ b_{nj} \end{pmatrix}$$

We can see that each element c_{ij} of matrix C is the result of n multiplications and $(n - 1)$ additions. The matrix multiplication is therefore the result of $(2n - 1)n^2$ arithmetic operations and the computational complexity is in $O(n^3)$.

Let's consider multiple ways to divide this into independent tasks:

- at the coarsest level, we could have a task that performs the entire matrix multiplication. There is no work division and therefore no performance gain when compared to the sequential way of doing it, unless we consider performing many matrices multiplications in parallel
- at the finest level, we could have each task perform the computation of a single c_{ij} element, that is, $(2n - 1)$ arithmetic operations, with a complexity in $O(n)$. These operations are extremely fast and will be measured in microseconds at worst. In this case, and unless n is very large, the overhead of parallelizing the tasks and distributing them over a grid will cost more than the actual computation
- a more appropriate level of granularity would be to have each task compute an entire column of the resulting matrix C:

$$\begin{pmatrix} c_{1j} \\ \dots \\ c_{nj} \end{pmatrix} = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \times \begin{pmatrix} b_{1j} \\ \dots \\ b_{nj} \end{pmatrix}$$

Each task will perform $(2n - 1)n$ arithmetic operations and its complexity is in $O(n^2)$. For sufficiently large values of n , executing the tasks in parallel will provide a noticeable performance gain. The greater the value of n , the higher the gain.

Conclusion: choosing the granularity of the tasks is a very important part of the design of a grid-enabled application. As a rule of thumb, if the execution of a task takes less than 10 milliseconds, you should consider coarser tasks or sequential execution. At the same time, remember that JPPF is good at executing workloads with many tasks. You have therefore to find a balance between the granularity of the tasks and the level of parallelization that the division of the workload provides.

2.4 Networking considerations

2.4.1 Two channels per connection

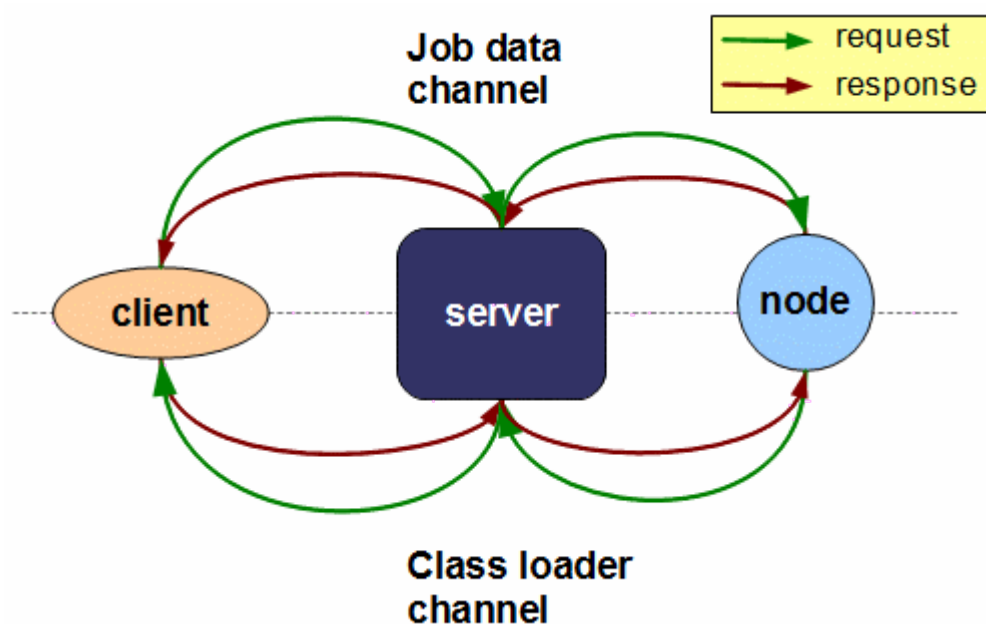
Each connection between a server and any other component is in fact a grouping of two network channels:

- 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.

2.4.2 Asynchronous networking

In JPPF, all network communications are asynchronous and follow a protocol based on a request/response paradigm, where multiple requests can be handled concurrently by each connection. 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 process multiple job concurrently; moreover, they can execute multiple tasks in parallel, which can be part of one or more jobs
- in the same way, a single client / server connection can process multiple job at a time; furthermore, 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, multiple jobs can be processed in parallel as well, since a server attaches to another server in exactly the same way as a node.

2.4.3 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
------	-----------------------------

2.4.4 Connection handshaking

When a connection to the JPPF server is established, the connecting component first sends a single integer value to the server, which allows the server to know what type of channel is connecting: node vs. client vs. other server, as well as job data vs. class loader vs. JMX vs. heartbeating channel. All known channel types are enumerated in the [JPPFIdentifier](#) class, which is internal and thus not exposed in the public API.

This technique allows all types of channel to connect to the same server port, with the benefit of simplifying both the JPPF grid configuration and the network security settings (for instance by opening a single server-side port in a firewall).

2.5 Sources of parallelism

2.5.1 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. Multiple jobs can be processed concurrently with a single connection, over multiple connections to the same driver (I/O parallelism), over multiple connections to different drivers, or any combination of these.

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.

2.5.2 At the server level

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

Number of attached nodes

The number of nodes in a JPPF grid is the most important factor of performance speed up. It determines how much of the overall workload can be effectively performed in parallel.

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 connections, 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 number 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. This pool grows and shrinks dynamically, based on the number of nodes connected to the server, and their activity. 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.

2.5.3 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).

3 Tutorial : A first taste of JPPF

3.1 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 1.7 or later and Apache Ant 1.8.0 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

3.2 Overview

3.2.1 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

3.2.2 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.

3.3 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 implementation of the interface [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.

3.4 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

3.4.1 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(String jobName) throws Exception {
    // create a JPPF job
    JPPFJob job = new JPPFJob();

    // give this job a readable name that we can use to monitor and manage it.
    job.setName(jobName);

    // 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.

3.4.2 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 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 in a `try-with-resource` block and provide it as a parameter for any method that needs it, in `TemplateApplicationRunner.java`.

It is always a good practice to release the resources used by the JPPF client when they are no longer used. Since [JPPFClient](#) implements [AutoCloseable](#), this can be done conveniently in a `try-with-resources` statement:

```
try (JPPFClient jppfClient = new JPPFClient()) {
    // ... use the JPPF client ...
}
```

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 and execute a blocking job
JPPFJob job = runner.executeBlockingJob(jppfClient);
```

As we can see, the job creation, its execution and the processing of its results are all encapsulated in a call to the method `executeBlockingJob(JPPFClient)`:

```

/**
 * Execute a job in blocking mode.
 * The application will be blocked until the job execution is complete.
 * @param jppfClient the {@link JPPFClient} instance which submits the job for execution.
 * @throws Exception if an error occurs while executing the job.
 */
public void executeBlockingJob(JPPFClient jppfClient) throws Exception {
    // Create a job
    JPPFJob job = createJob("Template blocking job");

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

    // process the results
    processExecutionResults(job.getName(), results);
}

```

The call to `createJob(jppfClient)` is exactly what we saw in the [previous section](#).

The next statement will send the job to the server and wait until it has been executed and the results are returned:

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

We can see that the results are returned as a list of Task 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 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 `Task.setThrowable(Throwable)` is called. JPPF considers that exception processing is part of the life cycle of a task and provides the means to capture this information accordingly.

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

```

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 `Task<E>` API provides two convenience methods to help doing this: `setResult(E)` and `getResult()`, however it is not mandatory to use them, and you can implement your own result handling scheme, or it could simply be a part of the task's design.

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

```

if (task.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.

3.5 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:

3.5.1 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: 6112, uuid: 4DC8135C-A22D-2545-E615-C06ABBF04065
management initialized and listening on port 11191
ClientClassServer initialized
NodeClassServer initialized
ClientJobServer initialized
NodeJobServer 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.

3.5.2 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, uuid: 4B7E4D22-BDA9-423F-415C-06F98F1C7B6F
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.

3.5.3 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 process id: 4780, uuid: 011B43B5-AE6B-87A6-C11E-0B2EBCFB9A89
[client: jppf_discovery-1-1 - ClassServer] Attempting connection to the class server ...
[client: jppf_discovery-1-1 - ClassServer] Reconnected to the class server
[client: jppf_discovery-1-1 - TaskServer] Attempting connection to the task server ...
[client: jppf_discovery-1-1 - TaskServer] Reconnected to the JPPF task server
Results for job 'Template blocking job' :
Template blocking job - Template task, 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.

3.6 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](#), 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:

```
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-x.y.z-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:

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

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

3.7 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

3.7.1 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 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.

3.7.1.1 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.

3.7.1.2 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:

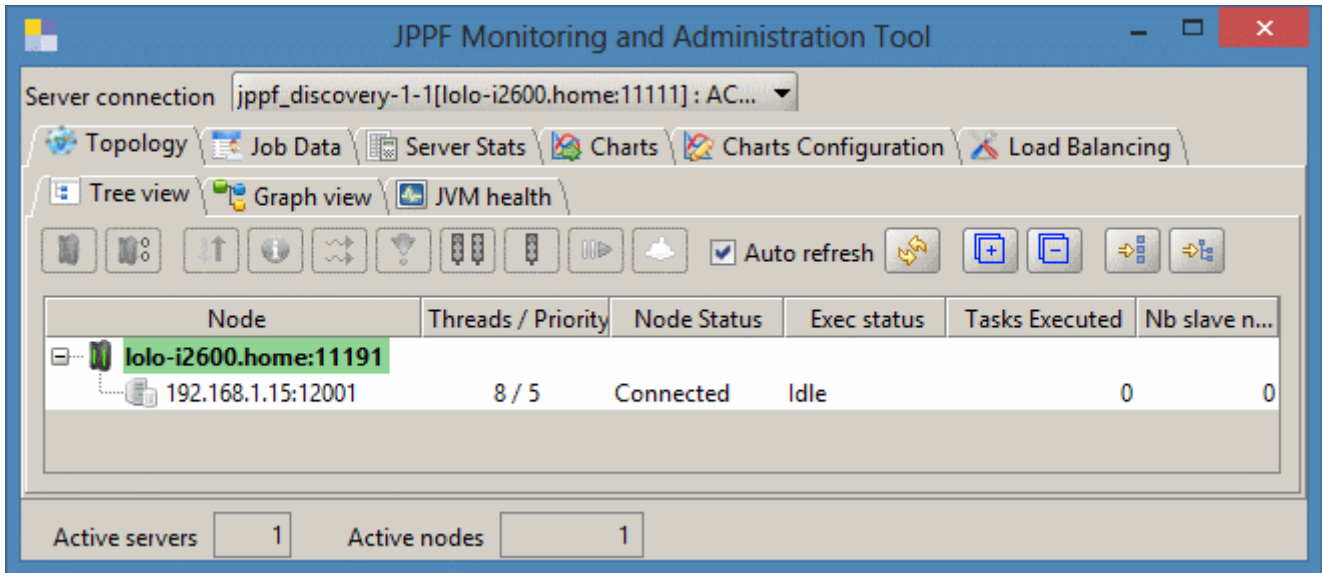
```

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.getSLA().setSuspended(true);
    return job;
}

```

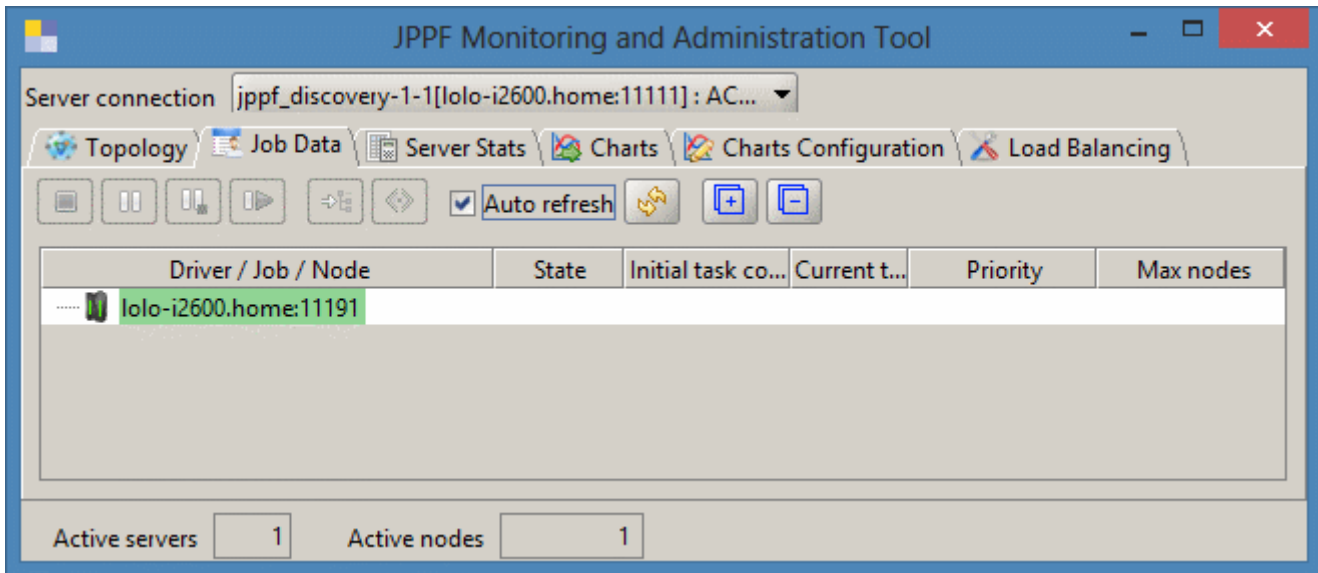
3.7.1.3 Step 3: start the JPPF components

If you have stopped the server and node, simply restart them as described in the first two steps of section 3.5 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.

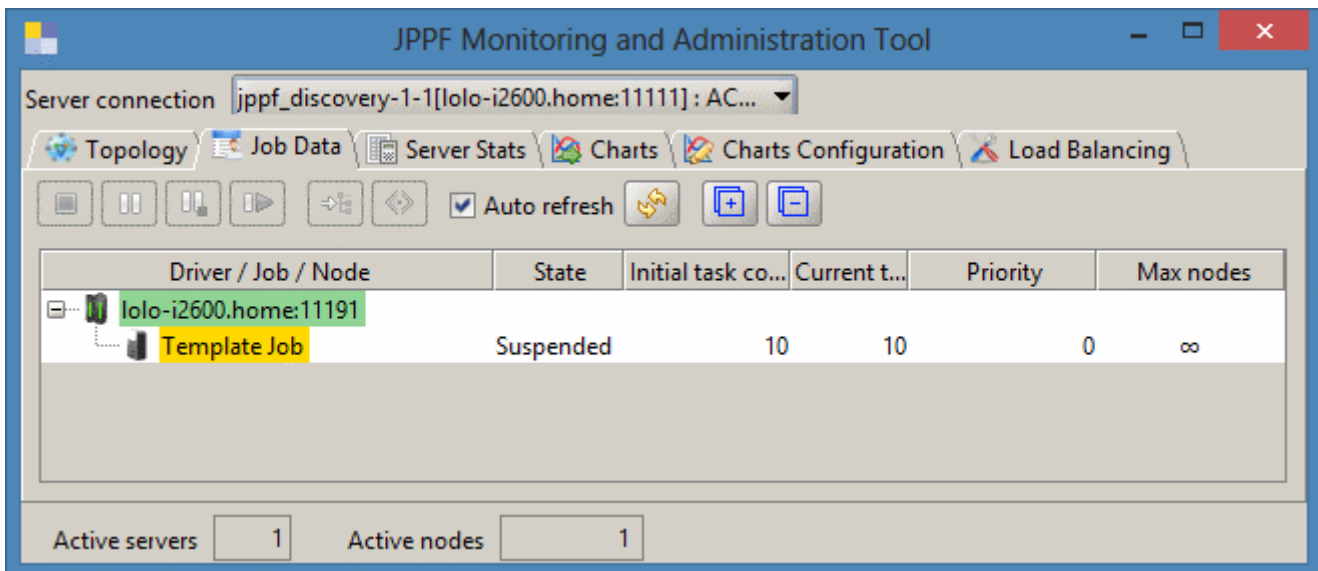
Let's switch to the "Job Data" panel, which should look like this:



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.

3.7.1.4 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 changes:

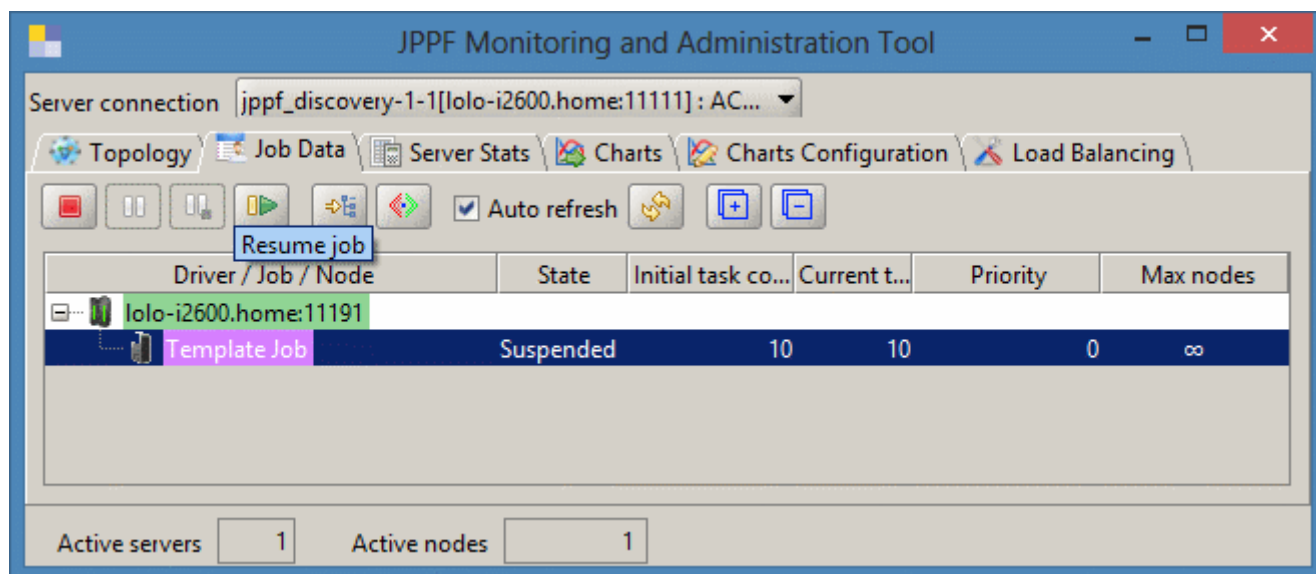


We now see that a job is present in the server's queue, in suspended state (yellow highlighting). Here is an explanation of the columns in the table:

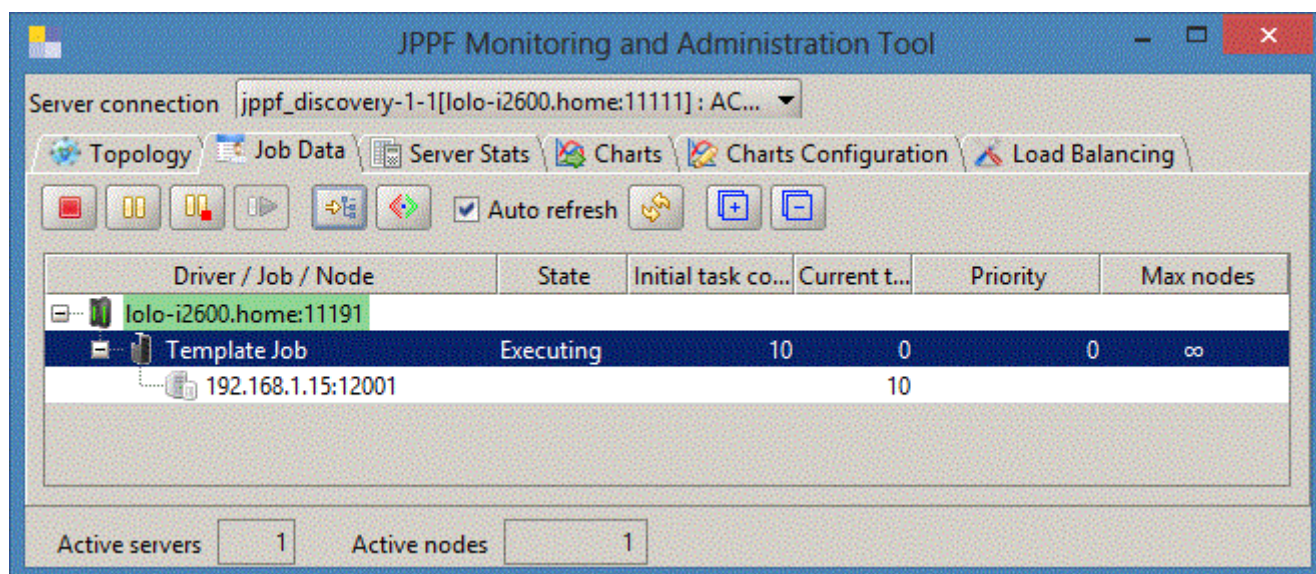
- "Driver / Job / Node" : displays an identifier for a server, for a job submitted to that server, or for a node to which some of the tasks in the job have been dispatched for execution
- "State" : the current state of a job, either "Suspended" or "Executing"
- "Initial task count" : the number of tasks 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

3.7.1.5 Step 5: resume 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...

3.8 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.

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 implementation 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> {  
    @Override  
    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> {
    @Override
    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 = ...;
Throwable t = task.getThrowable();
if (t != null) {
    t.printStackTrace();
}
```

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

For both methods, an attempt to cancel the task will be performed, by calling [Thread.interrupt\(\)](#) on the thread that executes it, *then* the onCancel() or onTimeout() method will be invoked. The implication is that the callback invocation takes place after the task's run() method returns, whether it was immediately interrupted (if the thread was doing an interruptible operation) or not.

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> {
    @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(String date, 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 Task<Object> setTimeoutSchedule(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 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 Getting information on the node executing the task

A [Task](#) can obtain information on the node using the following methods:

```
public interface Task<T> extends Runnable, Serializable {
    // is the task executing in a node or in the client?
    public boolean isInNode()
    // get the node executing this task, if any
    public Node getNode()
}
```

The `isInNode()` method determines whether the task is executing within a node or within a client with local execution enabled.

The `getNode()` method returns an instance of the [Node](#) interface, defined as follows:

```
public interface Node extends Runnable {
    // Get this node's UUID
    String getUuid();
    // Get the system information for this node
    JPPFSystemInformation getSystemInformation();
    // Determine whether this node is local to another component
    boolean isLocal();
    // Determine whether this node is running in offline mode
    boolean isOffline();
    // Determine whether this node is a 'master' node for the provisioning features
    boolean isMasterNode();
    // Determine whether this node is a 'slave' node for the provisioning features
    boolean isSlaveNode();
    // Determine whether this node can execute Net tasks
    boolean isDotnetCapable();
    // Determine whether this node is an Android node
    boolean isAndroid();
    // Get the JMX connector server associated with the node
    JMXServer getJmxServer() throws Exception;
    // Reset the current task class loader if any is present
    ClassLoader resetTaskClassLoader(Object...params);
}
```

Note that `Task.getNode()` will return `null` if the task is executing within a client local executor.

Here is an example usage:

```
public class MyTask extends AbstractTask<String> {
    @Override
    public void run() {
        if (isInNode()) {
            setResult("executing in remote node with uuid = " + getNode().getUuid());
        } else {
            setResult("executing in client-local executor");
        }
    }
}
```

4.1.4 Accessing the job

A task can access the job it is a part of, using its `getJob()` method, which returns a [JPPFDistributedJob](#) instance. This enables any executing task to access information on the job it belongs to: job uuid, name, metadata, service level agreement and number of tasks.

This method can be used as in this example:

```
public class MyTask extends AbstractTask<String> {
    @Override
    public void run() {
        JPPFDistributedJob job = this.getJob();
        String jobKey = job.getMetadata().getParameter("JOB_KEY");
        setResult(jobKey + "-" + this.getId());
    }
}
```


4.1.5 Executing code in the client from a task

The [Task](#) API provides a method that will allow a task to send code for execution in the client:

```
public interface Task<T> extends Runnable, Serializable {  
    // send a callable for execution on the client side  
    public <V> V compute(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  
        String callResult = compute(isInNode() ? new NodeCallable() : new ClientCallable());  
        // set the callable result as this task's result  
        setResult(callResult);  
    }  
  
    public static class NodeCallable implements JPPFCallable<String> {  
        @Override public String call() throws Exception {  
            return "executed in the NODE";  
        }  
    }  
  
    public static class ClientCallable implements JPPFCallable<String> {  
        @Override public String call() throws Exception {  
            return "executed in the CLIENT";  
        }  
    }  
}
```

4.1.6 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  
    Task<T> 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.

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

```
public class MyTaskJmxListener implements NotificationListener {  
    @Override  
    public synchronized void handleNotification(Notification notif, Object handback) {  
        // cast to the JPPF notification type, then get and print the user object  
        Object userObject = ((TaskExecutionNotification) notif).getUserData();  
        System.out.println("received notification with user object = " + userObject);  
        // determine who sent this notification  
        boolean userNotif = ((TaskExecutionNotification) notif).isUserNotification();  
        System.out.println("this notification was sent by the "  
            + (userNotif ? "user" : "JPPF node"));  
    }  
}
```

A local [TaskExecutionListener](#) would be like this:

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

    // task notification event sent by user code
    @Override 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”.

4.1.7 Resubmitting a task

The class [AbstractTask](#) also provides an API which allows a task to request that it be resubmitted by the server, instead of being sent back to the client as an execution result. This can prove very useful for instance when a task must absolutely complete successfully, but an error occurs during its execution. The API for this is defined as follows:

```
public abstract class AbstractTask<T> implements Task<T> {
    // Determine whether this task will be resubmitted by the server
    public boolean isResubmit()
    // Specify whether this task should be resubmitted by the server
    public Task<T> setResubmit(final boolean resubmit)

    // Get the maximum number of times a task can be resubmitted
    public int getMaxResubmits();
    // Set the maximum number of times a task can be resubmitted
    public Task<T> setMaxResubmits(int maxResubmits);
}
```

Note that the `resubmit` and `maxResubmits` attributes are *transient*, which means that upon when the task is executed in a remote node, they will be reset to their initial value of `false` and `-1`, respectively.

The maximum number of times a task can be resubmitted may be specified in two ways:

- in the job SLA via the [maxTaskResubmits](#) attribute
- with the task's `setMaxResubmits()` method; in this case any value ≥ 0 will override the job SLA's value

Finally, a task resubmission only works for tasks sent to a remote node, and will not work in the client's local executor.

4.1.8 JPPF-annotated tasks

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

Here is an example:

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


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

```
JPPFJob job = new JPPFJob();
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 previously, except for the callback methods `onCancel()` and `onTimeout()`.

Here is an example using an annotated constructor:

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

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

Another example using an annotated static method:

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

JPPFJob job = new JPPFJob();
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()`.

4.1.9 Runnable tasks

Classes that implement either [Runnable](#) or [JPPFRunnableTask](#) 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:

```
JPPFJob job = new JPPFJob();
// adding an anonymous runnable task
Task<?> task1 = job.add(new JPPFRunnableTask() {
    @Override
    public void run() {
        System.out.println("Hello!");
    }
});
// adding a runnable task as a lambda expression
Task<?> task2 = job.add(() -> System.out.println("Hello again!"));
```

The following rules apply to Runnable tasks:

- the class must be serializable (this is automatically the case with `JPPFRunnableTask`)
- 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 a previous section, except for the callback methods `onCancel()` and `onTimeout()`.

4.1.10 Callable tasks

In the same way as `Runnable` tasks, classes implementing either [Callable<V>](#) or [JPPFCallable<V>](#) can be directly used as tasks. In this case, the `call()` method will be used as the task's execution entry point, and its return value will be the task's result. Here's an example:

```
JPPFJob job = new JPPFJob();
// adding an anonymous callable task
Task<?> task1 = job.add(new JPPFCallable<String>() {
    @Override
    public String call() throws Exception {
        return "Hello!";
    }
});
// adding a callable task as a lambda expression
Task<?> task2 = job.add(() -> "Hello again!");
```

The following rules apply to `Callable` tasks:

- the `Callable` class must be serializable (this is automatically the case with `JPPFCallable`)
- 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 a previous section, except for the callback methods `onCancel()` and `onTimeout()`.

4.1.11 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 [JPPF-annotated tasks](#)), 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(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 previously, except for the callback methods `onCancel()` and `onTimeout()`.

4.1.12 Interruptibility

Sometimes, it is not desirable that the thread executing a task be interrupted upon a timeout or cancellation request. To control this behavior, a task should override its `isInterruptible()` method, as in this example:

```
public class MyTask extends AbstractTask<String> {
    @Override public void run() {
        // ...
    }

    @Override public boolean isInterruptible() {
        // this task can't be interrupted
        return false;
    }
}
```

Note that, by default, a task which does not override its `isInterruptible()` method *is* interruptible.

Tasks that do not extend [AbstractTask](#), such as `Callable`, `Runnable`, Pojo tasks or tasks annotated with `@JPPFRunnable`, will need to implement the [Interruptibility](#) interface to override the interruptible flag, as in this example:

```
public class MyCallable implements Callable<String>, Serializable, Interruptibility {
    @Override public String call() throws Exception {
        return "task result";
    }

    @Override public boolean isInterruptible() {
        return false; // NOT interruptible
    }
}
```

4.1.13 Cancellation handler

Tasks that need a callback invoked immediately upon cancellation, whether the thread interruption succeeded or not, can implement the [CancellationHandler](#) interface, defined as follows:

```
public interface CancellationHandler {
    // Invoked immediately when a task is cancelled
    void doCancelAction() throws Exception;
}
```

This can be used on its own, or in combination with the `onCancel()` method as in this example:

```
public class MyTask extends AbstractTask<String> implements CancellationHandler {
    private long start;

    @Override public void run() {
        start = System.nanoTime();
        // ... task processing ...
    }

    @Override public void doCancelAction() throws Exception {
        long elapsed = (System.nanoTime() - start) / 1_000_000L;
        System.out.println("doCancelAction() called after " + elapsed + " ms");
    }

    @Override public void onCancel() {

```

```

        long elapsed = (System.nanoTime() - start) / 1_000_000L;
        System.out.println("onCancel() called after " + elapsed + " ms");
    }
}

```

The interface applies to all types of tasks: tasks that extend [AbstractTask](#), pojo tasks, Cancellable and Runnable tasks, along with [@JPPFRunnable](#)-annotated tasks.

4.1.14 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 [AbstractTask](#), 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:

```

public abstract class CommandLineTask<T> extends AbstractTask<T> {
    // list of commands passed to the shell
    public List<String> getCommandList()
    public CommandLineTask<T> setCommandList(List<String> commandList)
    public CommandLineTask<T> setCommandList(String... commands)

    // set of environment variables
    public Map<String,String> getEnv()
    public CommandLineTask<T> setEnv(Map<String, String> env)

    // directory in which the command is executed
    String getStartDir()
    public CommandLineTask<T> setStartDir(String startDir)
}

```

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

```

public abstract class CommandLineTask<T> extends AbstractTask<T> {
    public CommandLineTask(Map<String, String> env, String startDir, String... commands)
    public 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
public 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
public 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:

```

public abstract class CommandLineTask<T> extends AbstractTask<T> {
    public boolean isCaptureOutput()
    public CommandLineTask<T> setCaptureOutput(boolean captureOutput)

    // corresponds to what is sent to System.out / stdout
    public String getErrorOutput()
    // corresponds to what is sent to System.err / stderr
    public 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
    @Override
    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);
        }
    }
}

```

4.1.15 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:

```

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 ScriptedTask<T> 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:

```
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 (`jppfTask.setResult()` statement), then this time the task result will be the same as the script result “Hello 2”.

4.1.16 Dependencies between tasks

It is possible to specify that a task has a dependency on one or more other tasks. The dependency relationship guarantees that the dependencies of a task will always be executed before the task itself. To enable dependencies, the task must implement the [TaskNode](#) interface, defined as follows:

```
public interface TaskNode<T> extends Task<T> {
    // Add a set of dependencies to this task
    TaskNode<T> dependsOn(Collection<TaskNode<?>> tasks)
        throws JPPFDependencyCycleException;

    // Add a set of dependencies to this task
    TaskNode<T> dependsOn(TaskNode<?>...tasks) throws JPPFDependencyCycleException;

    // Get the dependencies of this task, if any
    Collection<TaskNode<?>> getDependencies();

    // Get the tasks that depend on this task, if any
    Collection<TaskNode<?>> getDependants();

    // Determine whether this task has at least one dependency
    boolean hasDependency();

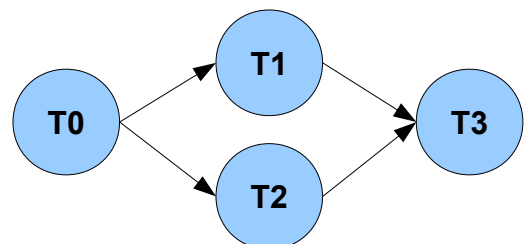
    // Determine whether this task has at least one other task that depends on it
    boolean hasDependant();
}
```

As we can see, this interface extends the [Task](#) interface and therefore represents a standard JPPF task, with extended capabilities. In practice, an implementation of a task with dependencies should extend the corresponding abstract class [AbstractTaskNode](#).

Dependencies are added using the `dependsOn(Collection<TaskNode<?>>)` and `dependsOn(TaskNode<?>...)` methods. In the example below, the code on the left creates the diamond dependency graph shown on the right:

```
public class MyTask extends AbstractTaskNode<String> {
    @Override
    public void run() {
        setResult("Hello!");
    }
}

MyTask t0 = new MyTask(), t1 = new MyTask(),
t2 = new MyTask(), t3 = new MyTask();
t0.dependsOn(t1, t2);
t1.dependsOn(t3);
t2.dependsOn(t3);
```



Cycles are checked for and detected when dependencies are added, that is, each time a `dependsOn(...)` method is invoked. When a cycle is detected in the dependency graph, a [JPPFDependencyCycleException](#) is raised, with a message describing how the cycle occurred.

Preventing cycles offers a guarantee that the tasks form a [directed acyclic graph](#) (DAG). Such a graph can be traversed in a [topological order](#), which ensures that no task is executed before its dependencies have completed.

For example, the following code creates a dependency cycle:

```
MyTask t1 = new MyTask("T1"), t2 = new MyTask("T2"), t3 = new MyTask("T3");  
t2.dependsOn(t3);  
t3.dependsOn(t1);  
t1.dependsOn(t2, t3);
```

It will result in an error similar to this:

```
org.jppf.node.protocol.graph.JPPFDependencyCycleException:  
  MyTask@7225790e (T1) ==> MyTask@35fb3008 (T2) ==> MyTask@3cbbc1e0 (T3)  
  ==> MyTask@7225790e (T1)
```

See also:

- [*adding tasks to a job*](#)
- [*dependency graph traversal*](#)

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 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 make creation of jobs easier.

4.2.1 Creating a job

To create a job, [JPPFJob](#) has a single no-arg constructor, which generates a unique universal identifier for the job:

```
public class JPPFJob extends AbstractJPPFJob<JPPFJob>
    implements Iterable<Task<?>>, Future<List<Task<?>>> {
    // creates a job with default values for its attributes
    public JPPFJob()

    // get the UUID of this job
    public String getUuid(uuid)
}
```

The job UUID is automatically generated as a pseudo-random string of hexadecimal characters in the standard 8-4-4-12 format. It can then be obtained with the job's `getUuid()` method.

Important note: *the role of the job UUID is critical, since it is used to distinguish the job from potentially many others in all JPPF grid topologies. It is also used in most job management and monitoring operations.*

Each job also has a name, which can be used to identify a job in a human readable way. When a job is created, its name is set to the job UUID. It can later be changed or accessed with the following accessors:

```
public class JPPFJob extends AbstractJPPFJob<JPPFJob>
    implements Iterable<Task<?>>, Future<List<Task<?>>> {
    // get the name of this job
    public String getName()

    // assign a name to this job
    public void setName(String name)
}
```

Note that the job's name is displayed in the "job data" view of the JPPF graphical administration console.

4.2.2 Adding tasks to a job

As we have seen in the [task objects](#) section about the various forms of tasks that we can use in JPPF, [JPPFJob](#) provides several methods to add tasks to a job.

4.2.2.1 Adding a standard JPPF task

```
public Task<?> add(Task<?> task) throws JPPFException
```

The return value is the task provided as input. If the task has already been added, then adding it again time has no effect.

If the task is an instance of [TaskNode](#) (if it has [dependencies](#)), then all the tasks in its dependency graph will also be added if not already present. This ensures that the job will not be submitted with an incomplete graph that would prevent its completion.

4.2.2.2 Adding an annotated, Runnable or Callable task

```
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 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:

```
Task<?> task = job.add(new MyRunnableTask());
MyRunnableTask runnableTask = (MyRunnableTask) task.getTaskObject();
```

As JPPF uses reflection to properly wrap the task, an eventual exception may be thrown, wrapped in a JPPFException.

4.2.2.3 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 optional args parameter 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 uses reflection to properly wrap the task, an eventual exception may be thrown, wrapped in a JPPFException.

4.2.3 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 ...
}
```

4.2.4 Job submission

Jobs are submitted with the [JPPFClient API](#), as seen later on in this manual. Job submission can be either synchronous, using the [JPPFClient.submit\(JPPFJob\)](#) method, or asynchronous, using the [JPPFClient.submitAsync\(JPPFJob\)](#) method.

This is illustrated in this example:

```

JPPFClient client = ...;
// a new job is blocking by default
JPPFJob blockingJob = new JPPFJob();
blockingJob.add(new MyTask());
// blocks until the job has completed
List<Task<?>> results = client.submit(blockingJob);

JPPFJob nonBlockingJob = new JPPFJob();
nonBlockingJob.add(new MyTask());
// returns immediately, without blocking the current thread
client.submitAsync(nonBlockingJob);
// ... later on, collect the results
List<Task<?>> results2 = nonBlockingJob.awaitResults();

```

4.2.5 Job execution results

[JPPFJob](#) provides the following methods to explore and obtain the execution results of its tasks:

```

public class JPPFJob extends AbstractJPPFJob<JPPFJob>
implements Iterable<Task<?>>, Future<List<Task<?>>> {
    // Get the count of tasks in this job that have completed
    public int executedTaskCount()
    // Get the count of tasks in this job that haven't yet been executed
    public int unexecutedTaskCount()
    // Wait until all execution results of the tasks in this job have been collected
    public List<Task<?>> awaitResults()
    // Wait until all execution results of the tasks in this job have been collected
    // or the timeout expires, whichever happens first
    public List<Task<?>> awaitResults(final long timeout)
    // Get the list of currently available task execution results
    public List<Task<?>> getAllResults()
    // Get the execution status of this job
    public JobStatus getStatus()
    // determine whether this job was cancelled
    public boolean isCancelled()
    // determine whether this job has completed normally or was cancelled
    public boolean isDone()
    // wait until the job is done
    public List<Task<?>> get() throws InterruptedException, ExecutionException
    // wait until the job is done or the timeout expires, whichever happens first
    public List<Task<?>> get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException
}

```

Note that the `awaitResults()` methods will block until the job has completed, or the timeout expires if any is specified. If the timeout expires, an incomplete list of results will be returned. By contrast, `getAllResults()` will return immediately with a partial list of task execution results, possibly empty if no result was received yet.

The `getStatus()` method returns an indication of the job's completion status, as one of the values defined in the [JobStatus](#) enum:

```

public enum JobStatus {
    // The job was just submitted
    SUBMITTED,
    // The job is currently in the submission queue (on the client side)
    PENDING,
    // The job is being executed
    EXECUTING,
    // The job execution is complete
    COMPLETE,
    // The job execution has failed
    FAILED
}

```

A notable difference between the `awaitResults(long)` and `get(long, TimeUnit)` methods is that the `get(...)` method will throw a `TimeoutException` whenever the specified timeout expires before the job completes. Other than that,, `awaitResults(timeout)` is equivalent to `get(timeout, TimeUnit.MILLISECONDS)`.

4.2.6 Cancelling a job

Cancelling a job can be performed with the `cancel()` and `cancel(boolean mayInterruptIfRunning)` methods of [JPPFJob](#). The `mayInterruptIfRunning` flag specifies whether the job can be cancelled while it is being executed: if the flag is `true` and the job is executing, then it will not be cancelled and the `cancel(...)` method will return `false`.

Note that the return value of `isCancelled()` will reflect the cancelled state of the job, but only if it was cancelled within the scope of the JPPF client application: with the [JPPFClient API](#), or `JPPFJob.cancel(boolean)`, or as the result of setting an [expiration schedule in the job's client SLA](#).

Example usage:

```
try (JPPFClient client = new JPPFClient()) {
    // create and populate the job
    JPPFJob job = new JPPFJob().setName("myJob");
    job.add(new MyTask());

    // submit the job asynchronously
    client.submitAsync(job);

    ...

    // cancel the job
    job.cancel(true);

    // get and process the job results
    List<Task<?>> results = job.awaitResults();
    ...
}
```

4.3 Jobs runtime behavior, recovery and failover

4.3.1 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.

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.

4.3.2 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 [DefaultFilePersistenceManager](#). 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:

```

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 [DefaultFilePersistenceManager](#) will use the serializations scheme configured for the client. Finally, this persistence manager is shown in action in the [Job Recovery](#) related sample.

4.3.3 Job lifecycle notifications: JobListener

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 with the job, defined as follows:

```

// 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:

```

// 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()

    // Whether the current job dispatch is sent to a remote driver
    public boolean isRemoteExecution()

    // Get the the connection used to send the job dispatch to a remote driver
    public JPPFClientConnection getConnection()
}

```

Note that the `getTasks()` method is only useful for `jobDispatched()` and `jobReturned()` notifications. In all other cases, it will return `null`. The same applies to the methods `isRemoteExecution()` and `getConnection()`.

Furthermore, `getConnection()` will also return `null` if `isRemoteExecution()` returns `false`, that is, if the job dispatch is executed in the client-local executor.

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

```
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. It can also be used to collect the results of non-blocking jobs in a fully asynchronous way.

If you do not need to implement all the methods of [JobListener](#), your implementation may instead extend the class [JobListenerAdapter](#), which provides an empty implementation of each method in the interface.

Multi-threaded usage note: *if you intend to use the same `JobListener` instance from multiple threads, for instance with multiple concurrent non-blocking jobs, you will need to explicitly synchronize the code of the listener.*

Here is a simple example of a thread-safe `JobListener` implementation:

```
// counts the total submitted and executed tasks for all jobs
public class MyJobListener extends JobListenerAdapter {
    private int totalSubmittedTasks = 0;
    private int totalExecutedTasks = 0;

    @Override
    public synchronized void jobStarted(JobEvent event) {
        JPPFJob job = event.getJob();
        // add the number of tasks in the job
        totalSubmittedTasks += job.getJobTasks().size();
        System.out.println("job started: submitted = " + totalSubmittedTasks +
            ", executed = " + totalExecutedTasks);
    }

    @Override
    public synchronized void jobReturned(JobEvent event) {
        List<Task<?>> tasks = event.getJobTasks();
        // add the number of task results received
        totalExecutedTasks += tasks.size();
        System.out.println("job returned: submitted = " + totalSubmittedTasks +
            ", executed = " + totalExecutedTasks);
    }
}
```

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 Task₁ referencing A₁, ... , Task_n referencing A_n. We can see that, if the shared object is very large, we will quickly face memory issues.

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

```
public interface DataProvider extends 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 defined in the [Metadata](#) 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 ...
    }
}
```

Note 1: the association of a data provider to each task is done automatically by JPPF and is totally transparent to the application.

Note 2: from each task's perspective, the data provider should be considered read-only. Modifications to the data provider such as adding or modifying values, will NOT be propagated beyond the scope of the node. Hence, a data provider cannot be used as a common data store for the tasks. Its only goal is to avoid excessive memory consumption and improve the performance of the job serialization.

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 share some common attributes, which the corresponding APIs reflect through inheritance.

A job SLA is represented by the class [JobSLA](#) for the server side SLA, and by the class [JobClientSLA](#) for the client side SLA. It can be accessed from a job using the related getters:

```
public abstract class AbstractJPPFJob<J> extends AbstractJPPFJob<J>>
    implements Serializable, JPPFDistributedJob

    // The job's server-side SLA
    public JobSLA getSLA()

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

Example usage:

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

Also note that both classes extend the common class [JobCommonSLA](#). We will detail them in the next 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](#) class, declared as follows:

```
public class JobCommonSLA<T> extends JobCommonSLA<T>> implements Serializable {
    ...
}
```

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:

```
// 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
JobSLA sla = job.getSLA().setExecutionPolicy(serverPolicy);
// set the client-side policy
JobClientSLA clientSla = job.getClientSLA().setExecutionPolicy(clientPolicy);
// print an XML representation of the server-side policy
System.out.println("server policy is:\n" + sla.getExecutionPolicy());
```

4.5.1.2 Preference policy

A preference policy defines an ordered set of execution policies, where eligible channels or nodes will be chosen from those that satisfy the policy with the foremost position.

For instance, if we have 3 policies P1, P2 and P3 ,in that order of preference, and the following occurs:

- 0 nodes satisfy policy P1
- 2 nodes satisfy policy P2
- 3 nodes satisfy policy P3

then eligible nodes will be chosen from those which satisfy the policy P2.

A preference policy is an instance of the [Preference](#) class, created with one of the constructors below:

```
public class Preference extends LogicalRule {
    // Create this preference with an array of policies
    public Preference(ExecutionPolicy... policies)
    // Create this preference with a list of policies
    public Preference(List<ExecutionPolicy> policies)
}
```

It can be obtained from or set onto a job SLA with the following accessors:

```
public class JobCommonSLA<T> extends JobCommonSLA<T>> implements Serializable {
    // Get the preference policy for this job SLA
    public Preference getPreferencePolicy()
    // Set the preference policy for this job SLA
    public T setPreferencePolicy(Preference preferencePolicy)
}
```

Performance implications: with n nodes, and p policies in the preference, the time complexity of evaluating the preference policy for a given job is in $O(np)$. Care should be taken not to abuse this feature, as it can significantly reduce the grid's overall performance.

Note 1: you may combine a preference policy with a "regular" execution policy in a job SLA, which will then further restrict the number of eligible channels or nodes, no matter which of the preferred policies applies.

Note 2: as long as none of the preferred policies in the preference applies, the job is not scheduled for execution

Note 3: when used as or within a regular execution policy, a preference policy will always evaluate to `true`, effectively making it "transparent".

Example usage:

```
// at least 4 cores
ExecutionPolicy corePolicy = new AtLeast("availableProcessors", 4);
// at least 1GB of heap
ExecutionPolicy heapPolicy = new AtLeast("totalMemory", 1024*1024*1024);
// prefer heap size over the number of cores
Preference pref = new Preference(heapPolicy, corePolicy);
// set the preference policy
JPPFJob job = ...;
job.getSLA().setPreferencePolicy(pref);
```

4.5.1.3 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:

```
public class JobCommonSLA<T> extends JobCommonSLA<T>> implements Serializable {
    // job start schedule
    public JPPFSchedule getJobSchedule()
    public T setJobSchedule(JPPFSchedule schedule)
    // job expiration schedule
    public JPPFSchedule getJobExpirationSchedule()
    public T setJobExpirationSchedule(JPPFSchedule schedule)
}
```

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

As a fixed length of time

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

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

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

As a specific date/time

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

The date format is specified as a pattern for a [SimpleDateFormat](#) instance. Here is an example usage::

```
JPPFJob myJob = new JPPFJob();
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);
// alternatively
schedule = new JPPFSchedule(ZonedDateTime.of(2010, 9, 30, 12, 8, 0, 0, ZoneId.of("Europe/Paris")));
myJob.getSLA().setJobExpirationSchedule(schedule);
```

4.5.1.4 Maximum dispatch size

When a job is dispatched to a driver by the client, or to a node by the driver, the number of tasks that are dispatched at once is normally determined by the [configured load-balancer](#). This behavior can be overridden by specifying an explicit maximum number of tasks for each dispatch of a job. Whenever the number of tasks computed by the load-balancer is higher than the one specified for the job, it is overridden. This is done in a client-side or server-side SLA with the following getter and setter:

```
public class JobCommonSLA<T> extends JobCommonSLA<T> implements Serializable {
    // Get the maximum number of tasks allowed in a dispatch of the job
    public int getMaxDispatchSize()

    // Set the maximum number of tasks allowed in a dispatch of the job
    public T setMaxDispatchSize(int maxDispatchSize)
}
```

By default, there is no limit to the dispatch size set in the SLA, and `getMaxDispatchSize()` returns [Integer.MAX_VALUE](#).

Example usage:

```
JPPFJob job = ...;
// send no more than 5 tasks at once to the server
job.getClientSLA().setMaxDispatchSize(5);
```

4.5.1.5 Allowing concurrent dispatches to the same channel

Since each connection of a client to a server, or from a node to a server, can handle multiple concurrent jobs, there can be situations where multiple dispatches of the same job will be processed concurrently by the driver or node. While this by no means is a problem for JPPF, an application may want to prevent this behavior from happening. This can be done by setting a specific flag with the following accessors:

```
public class JobCommonSLA<T> extends JobCommonSLA<T> implements Serializable {
    // Whether to allow concurrent dispatches of the job to the same channel
    public boolean isAllowMultipleDispatchesToSameChannel()

    // Specify whether to allow concurrent dispatches of a job to the same channel
    public T setAllowMultipleDispatchesToSameChannel(
        boolean allowMultipleDispatchesToSameChannel)
}
```

By default, multiple concurrent dispatches are enabled, that is, the `isAllowMultipleDispatchesToSameChannel()` method will return true, unless `setAllowMultipleDispatchesToSameChannel(false)` is called first.

Example usage:

```
JPPFJob job = ...;
// forbid concurrent dispatches to the same node
job.getSLA().setAllowMultipleDispatchesToSameChannel(false);
```

4.5.2 Server side SLA attributes

A server-side SLA is described by the [JobSLA](#) class, declared as:

```
public class JobSLA extends JobCommonSLA<JobSLA> {  
    ...  
}
```

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:

```
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 Maximum number of node provisioning groups

A node provisioning group designates a set of nodes made of one master node and its [provisioned slave nodes](#), if it has any. The SLA allows restricting a job execution to a maximum number of node provisioning groups. This SLA attribute is useful whenever you want to take advantage of the fact that, by definition, a master and its slave nodes all run on the same machine, for instance to exploit data locality properties. This attribute's default value of is [Integer.MAX_VALUE](#) ($2^{31}-1$).

Note that this attribute does not specifically restrict the total number of nodes the job can run on, since each master node can have any number of slaves. For this, you also need to set the [maximum number of nodes attribute](#). Additionally, this attribute has no effect on the selection of nodes that are neither master nor slave, such as [offline nodes](#).

Example usage:

```
JPPFJob job = new JPPFJob();  
// only execute on a single group of master/slaves at a time  
job.getSLA().setMaxNodeProvisioningGroups(1);  
// further restrict to only the slave nodes in the provisioning group  
job.getSLA().setExecutionPolicy(new Equal("jppf.node.provisioning.slave", true));
```

4.5.2.4 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);
```

4.5.2.5 Broadcast jobs

A broadcast job is a specific type of job, for which each task will 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 JobSLA 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.

4.5.2.6 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 on all nodes it was dispatched to, and remove the job from the server queue. This behavior can be disabled on a per-job basis, for example if you want the job to complete but do not need the execution results.

Example usage:

```
JPPFJob myJob = new JPPFJob();
myJob.getSLA().setCancelUponClientDisconnect(false);
```

4.5.2.7 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 individual 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);
```

4.5.2.8 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 notably used with offline nodes, to work around the fact that offline nodes do not have remote class loading capabilities. The class path attribute, by default empty but not null, is accessed with the following methods:

```
public class JobSLA extends JobCommonSLA<JobSLA> {
    // get / set the class path associated with the job
    public ClassPath getClassPath()
    public JobSLA 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(Location<?> location);
    ClassPath add(Location<?> sourceLocation, Location<?> targetLocation);
    ClassPath add(Location<?> sourceLocation, Location<?> targetLocation,
        boolean copyToExistingFile);
    // remove an element from this classpath
    ClassPath remove(ClassPathElement element);
    // 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 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.

The `add(...)` method which takes a boolean attribute `copyToExistingFile` allows you to specify whether the target location should be downloaded and/or copied from the source location, if it already exists on the node's file system. As the attribute name indicates, this only applies to target locations that are files, that is, either instances of [FileLocation](#) or [URLLocation](#) instances with a "file" URL protocol (e.g "file:/home/user/mylib.jar").

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 source (relative to the client) location of this element
    Location<?> getLocalLocation();
    // get the target (relative to the node) location of this element, if any
    Location<?> getRemoteLocation();
    // whether to copy to an already existing file target
    boolean isCopyToExistingFile();
    // 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 in 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(location);

// add another jar to download from Maven Central,
// which will be copied onto the node's local file system
Location<URL> source = new MavenCentralLocation("org.jpjpf:jppf-common:6.0");
Location<String> target = new FileLocation("templib/jppf-common-6.0.jar");
// don't download from maven central and copy to a file if the jar file already exists
classpath.add(source, target, false);

// tell the node to reset the tasks classloader with this new class path
classpath.setForceClassLoaderReset(true);
```

4.5.2.9 Maximum number of tasks resubmits

As we have seen in the “[resubmitting a task](#)” section, tasks have the ability to schedule themselves for resubmission by the server. The job server-side SLA allows you to set the maximum number of times this can occur, with the following accessors:

```
public class JobSLA extends JobCommonSLA<JobSLA> {
    // get the maximum number of times a task can resubmit itself
    // via AbstractTask.setResubmit(boolean)
    public int getMaxTaskResubmits()
    // set the maximum number of times a task can resubmit itself
    public JobSLA setMaxTaskResubmits(int maxResubmits)
    // Determine whether the max resubmits limit for tasks is also applied
    // when tasks are resubmitted due to a node error
    public boolean isApplyMaxResubmitsUponNodeError()
    // Specify whether the max resubmits limit for tasks should also be applied
    // when tasks are resubmitted due to a node error
    public JobSLA setApplyMaxResubmitsUponNodeError(boolean applyMaxResubmitsUponNodeError)
}
```

The default value for the `maxTaskResubmits` attribute is 1, which means that by default a task can resubmit itself at most once. Additionally, this attribute can be overridden by setting the [maxResubmits attribute](#) of individual tasks.

The `applyMaxResubmitsUponNodeError` flag is set to false by default. This means that, when the tasks are resubmitted due to a node connection error, the resubmit will not count with regards to the limit. To change this behavior, `setApplyMaxResubmitsUponNodeError(true)` must be called explicitly.

Example usage:

```
public class MyTask extends AbstractTask<String> {
    @Override public void run() {
        // unconditional resubmit could lead to an infinite loop
        setResubmit(true);
        // the result will only be kept after the max number of resubmits is reached
        setResult("success");
    }
}

JPPFJob job = new JPPFJob();
job.add(new MyTask());
// tasks can be resubmitted 4 times, meaning they can execute up to 5 times total
job.getSLA().setMaxTaskResubmits(4);
// resubmits due to node errors are also counted
job.getSLA().setApplyMaxResubmitsUponNodeError(true);
// ... submit the job and get the results ...
```

4.5.2.10 Disabling remote class loading during job execution

Jobs can specify whether remote class loader lookups are enabled during their execution in a remote node. When remote class loading is disabled, lookups are only performed in the local classpath of each class loader in the class loader hierarchy, and no remote resource requests are sent to the server or client. This is done with the following accessors:

```
public class JobSLA extends JobCommonSLA<JobSLA> {
    // Determine whether remote class loading is enabled for the job. Default to true
    public boolean isRemoteClassLoadingEnabled()

    // Specify whether remote class loading is enabled for the job
    public JobSLA setRemoteClassLoadingEnabled(boolean enabled)
}
```


Note 1: when remote class loading is disabled, the classes that the JPPF node normally loads from the server cannot be loaded remotely either. It is thus required to have these classes in the node's local classpath, which is usually done by adding the "jppf-server.jar" and "jppf-common.jar" files to the node's classpath.

Note 2: if a class is not found while remote class loading is disabled, it will remain not found, even if the next job specifies that remote class loading is enabled. This is due to the fact that the JPPF class loaders maintain a cache of classes not found to avoid unnecessary remote lookups. To avoid this behavior, the task class loader should be reset before the next job is executed.

Example usage:

```
JPPFJob job = new JPPFJob();
// disable remote class loading at execution time
job.getSLA().setRemoteClassLoadingEnabled(false);
```

4.5.2.11 Grid policy

Jobs can also specify an execution policy that will be evaluated against the server and the *totality* of its nodes, instead of just against individual nodes as for the SLA's [execution policy](#) attribute we saw earlier in this documentation.

This grid policy is defined as a normal execution policy with two differences:

- it is evaluated against the properties of the *server*
- it may include any number of [server global policies](#) that count the nodes matching a given *node* policy

This policy is accessible with the following setter and getter of the SLA:

```
public class JobSLA extends JobCommonSLA<JobSLA> {
    // Get the global grid execution policy
    public ExecutionPolicy getGridExecutionPolicy()
    // Set the global grid execution policy
    public JobSLA setGridExecutionPolicy(ExecutionPolicy policy)
}
```

For example, to express and set the policy "execute the job when the server has at least 2 GB of available heap memory and at least 3 nodes with more than 4 processing threads each", we would code something like this:

```
int GB = 1024*1024*1024; // 1 GB
JPPFJob job = new JPPFJob();
// evaluated against each node's properties
ExecutionPolicy nodePolicy = new MoreThan("jppf.processing.threads", 4);
// evaluated against the server's properties
ExecutionPolicy gridPolicy = new AtLeast("availableMemory", 2*GB)
    .and(new NodesMatching(Operator.MORE_THAN, 3, nodePolicy));
// set the grid policy onto the SLA
job.getSLA().setGridExecutionPolicy(gridPolicy);
```

4.5.2.12 Specifying the desired node configuration

It is possible for a job to specify the configuration of the nodes it needs to run on and force eligible nodes to update their configuration accordingly and restart for the configuration changes to take place. The specified configuration includes all existing JPPF properties, in particular "jppf.java.path" and "jppf.jvm.options", which allow specifying the JVM and its options for running the node after restart. It also includes any custom, application-defined property than can be expressed in a configuration file.

This is done with the following [JobSLA](#) methods:

```
public class JobSLA extends JobCommonSLA<JobSLA> {
    // Get the configuration of the node(s) this job should be executed on
    public JPPFNodeConfigSpec getDesiredNodeConfiguration()

    // Set the configuration of the node(s) this job should be executed on
    public JobSLA setDesiredNodeConfiguration(JPPFNodeConfigSpec nodeConfigurationSpec)
}
```

The desired node configuration is specified as a [JPPFNodeConfigSpec](#) object, defined as follows:

```
public class JPPFNodeConfigSpec implements Serializable {
    // Initialize this object with a desired configuration and a restart flag set to true
    public JPPFNodeConfigSpec(TypedProperties desiredConfiguration)
        throws IllegalArgumentException
```

```

// Initialize this object with a desired configuration and restart flag
public JPPFNodeConfigSpec(TypedProperties desiredConfiguration, boolean forceRestart)
    throws IllegalArgumentException

// Get the desired JPPF configuration of each node
public TypedProperties getConfiguration()

// Determine whether to force the restart of a node after reconfiguring it
public boolean isForceRestart()
}

```

The configuration attribute specifies the properties that will be overridden or added to the node configuration. In terms of node selection, the JPPF server will prioritize the nodes whose configuration most closely matches the desired one, by computing a similarity score which relies on the distances between the string values of the desired and actual properties. Only the properties specified in the configuration attribute are compared.

The forceRestart flag determines whether a node should be restarted when it matches exactly the desired configuration. If set to true, the nodes will always be restarted. Otherwise, nodes that exactly match the desired configuration will not be restarted.

It is important to note that this SLA attribute is evaluated in combination with the other attributes of the job SLA. In particular, it should not be confused with the [execution policy](#), which is used to first filter eligible nodes, whereas the desired node configuration is applied to eligible nodes and triggers a configuration change and restart in those nodes.

There are restrictions as to the kind of nodes that can be affected by this SLA attribute: since a configuration change and restart of the node is triggered, this can only be done with *manageable* nodes, which excludes offline nodes and Android nodes. Furthermore, it does not apply to server-local nodes, since the node restart would also cause the server to be restarted.

Lastly, it is strongly advised to use this SLA attribute in combination with the [maximum number of nodes](#) and a [job expiration](#): since the reconfiguration and restart is very disruptive for the nodes, it has a non-trivial impact on performance, so you might want to limit the number of nodes that are restarted. Also, between the request for the node reconfiguration and the time the node becomes available after restart, the server *reserves* the node for the specific job involved. Setting an expiration timeout on the job ensures that the node can be reused for other jobs, should anything wrong happen. In effect, the server will remove all reservations for this job whenever it is cancelled or expires.

Example usage:

```

JPPFJob job = new JPPFJob();
// define the desired node configuration properties
TypedProperties props = new TypedProperties()
    .set(JPPFProperties.JVM_OPTIONS, "-server -Xmx1g")
    .setInt("property.1", 123456)
    .setString("property.2", "abcdef");
// create the node config spec with restart only when the properties don't match
JPPFNodeConfigSpec desiredConfig = new JPPFNodeConfigSpec(props, false);
// set the corresponding SLA attribute
job.getSLA().setDesiredNodeConfiguration(desiredConfig);
// limit to 2 nodes max
job.getSLA().setMaxNodes(2);
// ensure the job expires after 10 minutes max
job.getSLA().setJobExpirationSchedule(new JPPFSchedule(10L*60L*1000L));

```

4.5.2.13 Specifying the job persistence

[Job persistence in the driver](#) is specified via the persistenceSpec attribute of the SLA:

```

public class JobSLA extends JobCommonSLA<JobSLA> {
    // Get the specification of the job persistence in the driver
    public PersistenceSpec getPersistenceSpec()
}

```

This attribute is an instance of the class [PersistenceSpec](#), defined as follows:

```

public class PersistenceSpec implements Serializable {
    // Determine whether the job is persisted in the driver. Defaults to false
    public boolean isPersistent()
    // Specify whether the job is persisted in the driver
    public PersistenceSpec setPersistent(boolean persistent)
    // Whether to automatically execute the persisted job upon restart. Defaults to false
    public boolean isAutoExecuteOnRestart()
}

```

```
// Specify whether the driver should automatically execute the job after a restart
public PersistenceSpec setAutoExecuteOnRestart(boolean autoExecuteOnRestart)
// Whether the job should be deleted from the store upon completion. Defaults to true
public boolean isDeleteOnCompletion()
// Determine whether the job should be deleted from the store upon completion
public PersistenceSpec setDeleteOnCompletion(boolean deleteOnCompletion)
}
```

Instances of this class manage three boolean flags:

- the **"persistent"** flag determines whether the job is persisted at all. By default, it is set to **false**.
- the **"delete on completion"** flag determines whether the job should be removed from the store when it completes. This flag is set to **true** by default.
- the **"auto execute on restart"** flag tells a driver that, upon restart, it should automatically resubmit the job's unexecuted tasks until the job completes. This flag is set to **false** by default.

The following example shows how we would configure a persistent job that should be automatically executed upon driver restart and deleted from the store upon completion:

```
JPPFJob job = new JPPFJob();
job.getSLA().getPersistenceSpec()
    .setPersistent(true)
    .setAutoExecuteOnRestart(true)
    .setDeleteOnCompletion(true);
```

4.5.2.14 Maximum driver depth

In a JPPF [multi-server topology](#), where JPPF servers can offload a part of their workload (i.e. jobs) to one another, it is possible to limit the number of peer servers a job can be sent to. Accessing and setting this limit is done with the following methods of a job SLA:

```
public class JobSLA extends JobCommonSLA<JobSLA> {
    // Determine how many drivers a job can traverse before being executed on a node
    public int getMaxDriverDepth()

    // Set how many drivers a job can traverse before being executed on a node
    public JobSLA setMaxDriverDepth(int maxDriverDepth)
}
```

By default, there is no limit and `getMaxDriverDepth()` will return [Integer.MAX_VALUE](#).

Example usage:

```
JPPFJob job = ...;
// limit to a single driver, which means that the
// job cannot be offloaded to another driver
job.getSLA().setMaxDriverDepth(1);
```

4.5.2.15 Job dependencies specification

JPPF jobs can have dependencies on other jobs and thus form dependency graphs. To define the dependencies of a job, along with how it is processed within a graph, you can use the `jobDependenciesSpec` attribute of the job's SLA, which is an instance of the [JobDependencySpec](#) class:

```
public class JobDependencySpec implements Serializable {
    // Get the application-defined id assigned to a job
    public String getId()

    // Set the dependency id for the current job
    public JobDependencySpec setId(String id)

    // Get the application-defined ids of the job's dependencies
    public List<String> getDependencies()

    // Add the specified dependencies of the job
    public JobDependencySpec addDependencies(String...dependencies)

    // Add the specified dependencies of the job
    public JobDependencySpec addDependencies(Collection<String> dependencies)

    // Determine whether the job is a root in the dependency graph it and its dependencies
    // should be removed from the graph upon completion. The default is true
    public boolean isGraphRoot()
}
```

```

// Specify whether the job is a root in the dependency graph, implying that it and its
// dependencies should be removed from the graph upon completion
public JobDependencySpec setGraphRoot(boolean graphRoot)

// Determine whether the job has at least one dependency
public boolean hasDependency()

// Determine whether cancelling the job triggers the cancellation of the jobs that depend on it
public boolean isCascadeCancellation()

// Specify whether cancelling the job triggers the cancellation of the jobs that depend on it
public JobDependencySpec setCascadeCancellation(boolean cascadeCancellation)
}

```

The `id` attribute is an arbitrary string which uniquely identifies the job within the dependency graph. It can be distinct from the job UUID, which allows specifying dependencies on jobs that do not exist yet, or even jobs submitted from a separate (and possibly remote) JPPF client.

Here is an example usage:

```

JPPFJob myJob = new JPPFJob().setName("MyJob");
JobDependencySpec spec = myJob.getSLA().getDependencySpec();
spec.setID("MyJob")
    .setGraphRoot(true)
    .addDependencies("MyJob_2", "MyJob_3");

```

Note: for full details on the processing of job dependency graphs, please read the [Job dependencies and job graphs](#) section of this documentation.

4.5.3 Client side SLA attributes

A client-side SLA is described by the interface [JobClientSLA](#), defined as:

```

public class JobClientSLA extends JobCommonSLA<JobClientSLA> {
    ...
}

```

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.

The accessors for the attribute are defined as follows:

```

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

```

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);

```

4.5.3.2 Dependency graph traversal

When tasks in a job have [dependencies](#), there are two ways the graph they form can be executed:

Graph traversal on the server side: in this case, the entire job with all its tasks will be sent at once to the server. The graph traversal will be performed by the server, in a [topological order](#) of the tasks dependencies. Only the tasks that no longer have unrealized dependencies will be dispatched to the nodes for execution. **This is the default mode.**

Graph traversal on the client side: in this mode, only the tasks with unrealized dependencies will be sent to the server, where they can be further distributed between the nodes. This mode is mostly useful if you need to execute your jobs in a [client-local executor](#). Other than that, it is not recommended, as it may prevent a proper job recovery, especially if you rely on [jobs persistence](#).

Note: both modes imply that the distribution of a job with task dependencies will bypass the load-balancer settings, in order to guarantee the integrity of the dependency graph.

The type of graph traversal is set and accessed with the `graphTraversalInClient` attribute of the client SLA:

```
public class JobClientSLA extends JobCommonSLA<JobClientSLA> {  
    // Determine whether the traversal of the graph of tasks, if any, will occur on the client side  
    public boolean isGraphTraversalInClient()  
  
    // Specify whether the traversal of the graph of tasks, if any, will occur on the client side  
    public void setGraphTraversalInClient(boolean graphTraversalInClient)  
}
```

Example usage:

```
JPPFJob job = new JPPFJob();  
// set the graph traversal on the client side for this job  
job.getClientSLA().setGraphTraversalInClient(true);
```

See also:

- [dependencies between tasks](#)
- [adding tasks to a job](#)

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  
    Metadata setParameter(Object key, T value)  
    // Retrieve a parameter in the metadata  
    <T> T getParameter(Object key)  
    // Retrieve a parameter in the metadata  
    <T> T getParameter(Object key, T defaultValue)  
    // Remove a parameter from the metadata  
    <T> T 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();  
metaData  
    // set the memory footprint of each task to 10 KB  
    .setParameter("task.footprint", "" + (10 * 1024))  
    // set the duration of each task to 80 milliseconds  
    .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
- Operating system information, such as OS name, version and architecture, physical RAM, swap space, CPU load

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 [ExecutionPolicy](#). It can be built in 2 ways:

By API, using the classes in the [org.jppf.node.policy](#) package.

Example:

```
// define a policy allowing only nodes with 2 processing threads or more
ExecutionPolicy atLeast2ThreadsPolicy = new AtLeast("jppf.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("jppf.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>jppf.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="https://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.

JPPF supports any [JSR-223](#) / [javax.script](#) compliant script engine.

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 dispatched 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 it 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 Execution policy context

Each execution policy has access to a set of contextual information during its evaluation. This information pertains to the job against which the policy is evaluated, along with a snapshot of the server statistics (for server-side policies) at the time of the evaluation. This context is available with the method [ExecutionPolicy.getContext\(\)](#) and returns a [PolicyContext](#) object, defined as follows:

```

public class PolicyContext {
    // Get the job server side SLA, set at runtime by the server
    public JobSLA getSLA()

    // Get the job client side SLA, set at runtime by the server
    public JobClientSLA getClientSLA()

    // Get the job metadata, set at runtime by the server
    public JobMetadata getMetadata()

    // Get the number of nodes the job is already dispatched to
    public int getJobDispatches()

    // Get the server statistics
    public JPPFStatistics getStats()
}

```

Note that, depending on where the execution policy is evaluated, some parts of the context may not be available:

- when the policy is evaluated in a client local executor, the server statistics are not available
- when the policy is evaluated on by server side scheduler, the client-side SLA is not available
- when the policy is evaluated via a call to any method of [JPPFDriverAdminMBean](#) which takes a `NodeSelector` or `ExecutionPolicy` parameter, only the server statistics are available.

The context is available to any execution policy, however it will be especially useful for custom policies. For scripted policies, the elements of information it provides are already split into separate [variable bindings](#). Furthermore, keep in mind that the policy context is only valid in the scope of the policy's [accepts\(\)](#) method.

4.7.4 Custom policies

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

- the custom policy class must extend [CustomPolicy](#)
- the custom policy class must be deployed in the JPPF server classpath as well as the client's

Here is a sample custom policy code:

```
package mypackage;
import org.jppf.utils.PropertiesCollection;
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(PropertiesCollection 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 policy becomes then:

```
public class MyCustomPolicy extends CustomPolicy {
    public MyCustomPolicy(String...args) { super(args); }

    @Override public boolean accepts(PropertiesCollection 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) { }
        return n >= param; // node is accepted only if number of threads >= 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 add the corresponding class(es) to both the server's and the client's classpath, within either a jar file or a class folder.

4.7.5 Server global policies

JPPF 5.2 introduced a new type of execution policy which applies globally to *all* the nodes connected to a given server. It allows expressing rules such as *"execute a job only if there are more than 4 nodes, each with at least 2 processors and at least 4 GB of memory"*.

This type of execution policy is represented by the class [NodesMatching](#), defined as follows:

```
public class NodesMatching extends ExecutionPolicy {
    // Initialize this execution policy
    public NodesMatching(Operator operator, long expectedNodes, ExecutionPolicy nodePolicy)

    // Evaluate this policy against the nodes
    public boolean accepts(PropertiesCollection info)
}
```

The interesting part is in the constructor, which takes the following parameters:

- **operator**: one of the possible comparison operators defined in the [Operator](#) enum, that is, one of EQUAL, NOT_EQUAL, LESS_THAN, MORE_THAN, AT_LEAST, AT_MOST.
- **expectedNodes**: this is the number of nodes expected to match the comparison with the actual number of nodes that satisfy the **nodePolicy** parameter
- **nodePolicy**: an execution policy that will be evaluated against all the nodes. The number of nodes matching this policy will be compared to the **expectedNodes** parameter using the **operator** parameter as a comparison operator.

Note: it is important to remember that a [NodesMatching](#) policy is evaluated against the server, whereas the node policy in its constructor is evaluated against each individual node.

As an example, we would write the policy expressed above as follows:

```
// 1 GB = 1,073,741,824 bytes
long GB = 1024*1024*1024;
// node with at least 2 processors and at least 4 GB of heap
ExecutionPolicy nodePolicy = new AtLeast("availableProcessors", 2)
    .and(new AtLeast("maxMemory", 4*GB));
// more than 4 nodes satisfying the node policy
ExecutionPolicy globalPolicy = new NodesMatching(Operator.MORE_THAN, 4, nodePolicy);
```

Alternatively, it can also be written as an XML document:

```
<!-- more than 4 nodes such that: -->
<NodesMatching operator="MORE_THAN" expected="4">
    <AND>
        <!-- each node has at least 2 processors -->
        <AtLeast valueType="numeric">
            <Property>availableProcessors</Property>
            <Value>2</Value>
        </AtLeast>
        <!-- each node has at least 4 GB of heap -->
        <AtLeast>
            <Property>maxMemory</Property>
            <Value>4294967296</Value>
        </AtLeast>
    </AND>
</NodesMatching>
```

4.7.6 Execution policy arguments as expressions

Since JPPF 6.0, the arguments of most execution policies can also be expressed as the values of JPPF properties including [property substitutions](#) and [scripted expressions](#).

As an example, let's imagine the configuration of a node contains the following properties:

```
int.1 = 1
int.2 = 2
int.3 = 3
```

Now let's say we define an execution policy where jobs only execute on nodes that have at least 4 processing threads:

```
ExecutionPolicy policy = new AtLeast("jppf.processing.threads", 4);
```

Further, let's say that the expected number of processing threads is not static, and actually depends on the node's configuration. We could then rewrite the policy as in this example:

```
ExecutionPolicy policy = new AtLeast("jppf.processing.threads",
    "$script{ ${int.1} + ${int.3} }$");
```

Given the node configuration above, we can see that the value of the expression will resolve to 4 for this node.

We distinguish two cases, depending on the semantic of the argument:

1) Property name argument: this argument is always the first in an execution policy's constructor. It can now be either:

- a literal which represents the name of a property. This preserve the behavior from before JPPF 6.0
- or an expression that resolves to a value of the type handled by the execution policy

The limitation here is that this argument can never be a literal value, since any literal is interpreted as the name of a property. If a literal value is required, it may be specified instead in one of the right-side arguments (if any), or as a scripted expression, for instance: "\$script{4}\$".

In the example above, we used a property name expressed as the literal "jppf.processing.threads". We could also write it as a completely unrelated expression, for instance:

```
ExecutionPolicy policy = new AtLeast("$script{ 2 * ${int.3} }$",  
    "$script{ ${int.1} + ${int.3} }$");
```

For the node we considered, this policy would resolve to the comparison "6 >= 4".

2) Right-side argument(s): as seen in the example above, the arguments on the right side can be expressions that must resolve to the expected type of the argument, or to a literal of the expected type. For example, the policies below resolve to the same comparison "valueOf("jppf.processing.threads") >= 4" when evaluated against our node:

```
ExecutionPolicy p1 = new AtLeast("jppf.processing.threads", "$script{ 2 * ${int.2} }$");  
ExecutionPolicy p2 = new AtLeast("jppf.processing.threads", "4");  
ExecutionPolicy p3 = new AtLeast("jppf.processing.threads", 4);
```

Important: all scripted expressions have access to a predefined variable named "jppfSystemInfo" which references an object of type [JPPFSystemInformation](#).

Note 1: support for arguments as expressions is clearly documented in each execution policy's [javadoc](#)

Note 2: remember that in an expression, property substitutions are evaluated **first**, and scripted expressions are evaluated after that.

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`:

```
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 :

```
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 configuration
    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 two methods, depending on whether you want to submit the job synchronously or asynchronously :

```
public class JPPFClient extends AbstractGenericClient {
    // Submit a job synchronously and wait for the results
    public List<Tasks<?>> submit(JPPFJob job)

    // Submit a job asynchronously and return immediately
    public JPPFJob submitAsync(JPPFJob job)
}
```

These methods have different behaviors:

- the `submit()` method blocks until the job execution is complete. The return value is a list of tasks with their results, in the same order as the tasks that were added to the job.
- the `submitAsync()` method returns immediately, not waiting for the job to execute. It is up to the developer to collect the execution results, generally via the [JPPF job API](#).

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:

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

4.8.5 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.

4.8.6 Registering additional class loaders to handle requests from the driver

In some unusual use cases, it may be necessary to register additional class loaders with the client for a given job. This can happen if some of the tasks in the job are loaded with different class loaders, or in some situations when the job is submitted from a task executing in a node. To enable this, the super class of JPPFClient, [AbstractGenericClient](#), provides the following method:

```
public class AbstractGenericClient extends AbstractJPPFClient {
    // register a class loader for the specified job uuid
    public ClassLoader registerClassLoader(ClassLoader cl, String uuid)
}
```

The class loader must be registered *before* the job is submitted, otherwise it will have no effect. Additionally, the unregistration of the class loader is automatically performed by the JPPF client, once the job has completed.

Example usage:

```
JPPFClient client = new JPPFClient();
JPPFJob job = newJPPFJob();
// let's assume MyTask is loaded from a separate class loader
MyTask myTask = ...;
job.add(task);
ClassLoader cl = myTask.getClass().getClassLoader();
// register the class loader for the job, before submitting it
client.registerClassLoader(cl, job.getUuid());
List<Task<?>> result = client.submit(job);
```

4.8.7 Changing and retrieving the load-balancer settings

The load-balancer configuration can be dynamically updated and retrieved, even while the client is executing jobs, using the following methods in [JPPFClient](#):

```
public class JPPFClient extends AbstractGenericClient {
    // Get the current load-balancer settings
    public LoadBalancingInformation getLoadBalancerSettings()
    // Change the load balancer settings
    public void setLoadBalancerSettings(String algo, Properties params) throws Exception
}
```

The parameters of the `setLoadBalancerSettings()` method are those of the load-balancing algorithm and must not be prefixed. The `getLoadBalancerSettings()` method provides the current configuration encapsulated in a [LoadBalancingInformation](#) object.

Example usage:

```
JPPFClient jppfClient = new JPPFClient();
// configure the "proportional" load-balancer algorithm
TypedProperties props = new TypedProperties().setInt("initialSize", 5)
    .setInt("proportionalityFactor", 1);
jppfClient.setLoadBalancerSettings("proportional", props);
// retrieve the load-balancer configuration
LoadBalancingInformation lbi = jppfClient.getLoadBalancerSettings();
System.out.println("current algorithm is " + lbi.getAlgorithm());
System.out.println("algorithm parameters: " + lbi.getParameters());
```

4.8.8 Default job execution policies

A [JPPFClient](#) instance can have a default client and/or server side [execution policy](#). The default policies are applied only to submitted jobs that don't have an execution policy (i.e. the execution policy is null). Default execution policies can be set or retrieved with the following methods:

```
public class JPPFClient extends AbstractGenericClient {
    // Get the default server-side job execution policy
    public ExecutionPolicy getDefaultPolicy()
    // Set the default server-side job execution policy
    public void setDefaultPolicy(ExecutionPolicy defaultServerPolicy)
    // Get the default client-side job execution policy
    public ExecutionPolicy getDefaultClientPolicy()
    // Set the default client-side job execution policy
    public void setDefaultClientPolicy(ExecutionPolicy defaultClientPolicy)
}
```

Note: default policies can also be set within the JPPF client configuration, as [documented here](#).

Here is an example usage:

```
try (JPPFClient client = new JPPFClient()) {
    // by default, jobs can only execute on nodes with at least 4 CPUs
    client.setDefaultPolicy(new AtLeast("availableProcessors", 4));
    // by default, jobs cannot execute in the client local executor
    client.setDefaultClientPolicy(new Equal("jppf.channel.local", false));
}
```

4.8.9 Inspecting the jobs queue

You can list and count the jobs queued in a [JPPFClient](#) instance, using the following methods:

```
public class JPPFClient extends AbstractGenericClient {
    // Get the list of currently queued jobs
    public List<JPPFJob> getQueuedJobs()

    // Get a list of currently queued jobs, filtered by a JobSelector
    public List<JPPFJob> getQueuedJobs(JobSelector selector)

    // Get the current number of queued jobs
    public int getQueuedJobsCount()

    // Get the current number of jobs that satisfy a job selector
    public int getQueuedJobsCount(JobSelector selector)
}
```

Note that you can view or count either all jobs or jobs filtered by a [job selector](#).

Example usage:

```
JPPFJob job = new JPPFJob();
job.setBlocking(false);
job.add(new MyTask());

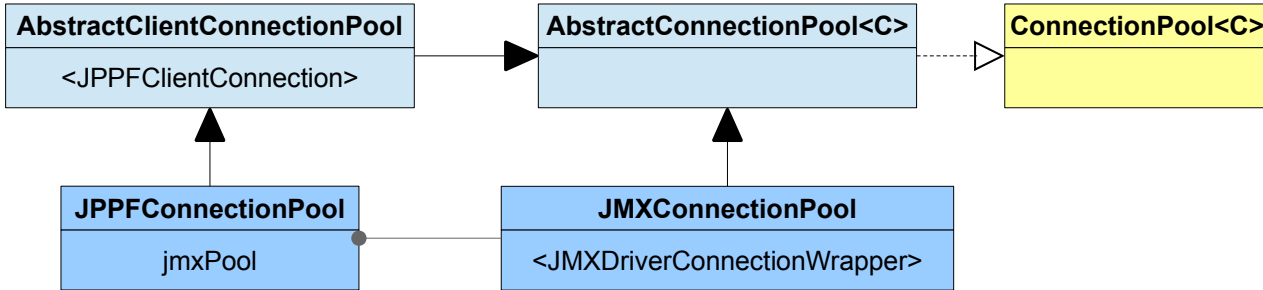
try (JPPFClient client = new JPPFClient()) {
    client.submitJob(job);
    List<JPPFJob> jobs = client.getQueuedJobs();
    // ... do something with the list of queued jobs ..
    final List<Task<?>> results = job.awaitResults();
    // ... process the results ...
}
```

4.9 Connection pools

All server connections in a JPPF client are organized into connection pools, whose number is determined by the client configuration properties. Their size is also based on the configuration and can also be changed dynamically via the JPPF APIs. In the next sections, we will see how connection pools can be configured, explored and programmatically accessed.

4.9.1 Connection pools class hierarchy

The class hierarchy for the client-side JPPF connections is as follows:



4.9.2 The JPPFConnectionPool API

The [ConnectionPool](#) interface and its direct abstract implementation [AbstractConnectionPool](#) provide the base methods to explore and access the connections in the pool:

```
public interface ConnectionPool<E extends AutoCloseable>
    extends Iterable<E>, AutoCloseable {
    // Get the next connection that is connected and available
    E getConnection();
    // Determine whether this pool is empty
    boolean isEmpty();
    // Get the current size of this pool
    int connectionCount();
    // Get the maximum size of this connection pool
    int getSize();
    // Set the maximum size of this pool, starting or stopping connections as needed
    int setSize(int size);
    // Get a list of connections held by this pool
    List<E> getConnections();
}
```

[AbstractClientConnectionPool](#) has the accessors to attributes specific to JPPF client connections to a driver:

```
public abstract class AbstractClientConnectionPool
    extends AbstractConnectionPool<JPPFClientConnection> {
    // Get the number of connections in this pool that have one of the specified statuses
    public int connectionCount(JPPFClientConnectionStatus...statuses)
    // Get a list of connections whose status is one of the specified statuses
    public List<JPPFClientConnection> getConnections(JPPFClientConnectionStatus...statuses)
    // Get the id of this pool
    public int getId()
    // Get the priority associated with this pool
    public int getPriority()
    // Check whether this pool is for SSL connections
    public boolean isSslEnabled()
    // Get the name of this pool.
    public String getName()
    // Get the uuid of the driver to which connections in this pool are connected
    public String getDriverUuid()
    // Get the host name of the remote driver
    public String getDriverHost()
    // Get the ip address of the remote driver
    public String getDriverIPAddress()
    // Get the port to use on the remote driver
    public int getDriverPort()
    // Get the JPPF client which holds this pool
    public JPPFClient getClient()
    // Get the driver's system information
    public JPPFSystemInformation getSystemInfo()
    // Get the jmx port to use on the remote driver
    public int getJmxPort()
    // Get the maximum number of jobs each connection can process concurrently
```

```

public int getMaxJobs()
// Set the maximum number of jobs each connection can process concurrently
public int setMaxJobs(int maxJobs)
}

```

The name of the pool, accessed via the `getName()` method, can be computed in different ways:

- when the pool is created from a [driver discovery plugin](#), its name is provided directly
- when UDP discovery is disabled, the pools names are read from the `jppf.drivers` configuration property:
`jppf.drivers = <driver_name_1> ... <driver_name_N>`
- when UDP discovery is enabled, the names are computed automatically in the form `jppf_discovery-n`, where *n* is the order of discovery of the pool, without any other meaning

The same mechanism applies to `getDriverHost()`, `getDriverPort()`, `getPriority()`, `isSslEnabled()`, `getMaxJobs()` and `getSize()`. Note that some of these attributes also have a setter method, which means that they can be changed dynamically y API.

The pool's actual size can be grown or shrunk dynamically, using the `setSize(int)` method. The JPPF client will create or close connections accordingly. An attempt to set a size equal to the current size will have no effect whatsoever. In some cases, when trying to reduce the connection pool's size, there may be too many connections in the pool that are busy executing jobs and the client will not be able to close all the requested connections. In this case, `setSize()` will return the new actual size, which may be smaller than the requested size.

Similarly, the number of jobs each connection can handle concurrently can be dynamically changed by calling `setMaxJobs(int)`. By default, this number of jobs is set to [Integer.MAX_VALUE](#), that is, $2^{31} - 1$, or 2,147,483,647.

4.9.3 Client connections

4.9.3.1 The *JPPFClientConnection* interface

As we have seen, connection pools contain and manage a set of connections from a client to a driver. These connections are represented by the [JPPFClientConnection](#) interface, defined as follows:

```

public interface JPPFClientConnection
extends ClientConnectionStatusHandler, AutoCloseable {
// Get the priority assigned to this connection
int getPriority();
// Shutdown this connection and release all the resources it is using
void close();
// Determine whether this connection was closed
boolean isClosed();
// Get the name assigned to this client connection
String getName();
// Determines if this connection is over SSL
boolean isSslEnabled();
// Get the driver's host name or ip address
String getHost();
// Get the port number on which the dirver is listeneing for connections
int getPort();
// Get the unique identifier of the remote driver
String getDriverUuid();
// Get the system information for the remote driver this connection refers to
JPPFSystemInformation getSystemInfo();
// Get the unique ID for this connection and its two channels
String getConnectionUuid();
// Get the pool this connection belongs to
JPPFConnectionPool getConnectionPool();
}

```

Note that most of these methods, except for `close()`, `isClosed()`, `getConnectionUuid()` and `getName()`, actually delegate to the [JPPFConnectionPool](#) to which the connection belongs. For example, `connection.getPort()` is equivalent to `connection.getPool().getDriverPort()`;

4.9.3.2 Status notifications for existing connections

Each 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, FAILED or CLOSED.

[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 implements 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);
    }
}

```

4.9.4 Exploring the connections in a pool

The [JPPFConnectionPool](#) class has higher-level methods to explore the connections with filtering and wait for them to be in a specified state:

```

public class JPPFConnectionPool extends AbstractClientConnectionPool {
    // Wait for the specified number of connections to be in the ACTIVE status
    public List<JPPFClientConnection> awaitActiveConnections(
        ComparisonOperator operator, int nbConnections)
    // Wait for the a connection to be in the ACTIVE state
    public JPPFClientConnection awaitActiveConnection()
    // Wait for the specified number of connections to be in the ACTIVE or EXECUTING state
    public List<JPPFClientConnection> awaitWorkingConnections(
        ComparisonOperator operator, int nbConnections)
    // Wait for a connection to be in the ACTIVE or EXECUTING} state
    public JPPFClientConnection awaitWorkingConnection()
    // Wait for the given number of connections to be in one of the given states
    public List<JPPFClientConnection> awaitConnections(ComparisonOperator operator,
        int nbConnections, JPPFClientConnectionStatus...statuses)
    // Wait for a connection to be in one of the specified states
    public JPPFClientConnection awaitConnection(JPPFClientConnectionStatus...statuses)
    // Wait for the given number of connections to be in one of the given states,
    // or for the given timeout to expire, whichever happens first
    public List<JPPFClientConnection> awaitConnections(ComparisonOperator operator,
        int nbConnections, long timeout, JPPFClientConnectionStatus...statuses)
}

```

The [ComparisonOperator](#) parameter is typically, but not necessarily, an element of the [Operator](#) enum, which defines the most common conditions that the number of connections must satisfy:

```

public enum Operator implements ComparisonOperator {
    // The number of connections is equal to the expected number
    EQUAL,
    // The number of connections is different from the expected number
    NOT_EQUAL,
    // The number of connections is at least to the expected number
    AT_LEAST,
    // The number of connections is at most the expected number
    AT_MOST,
    // The number of connections is strictly greater than the expected number
    MORE_THAN,
    // The number of connections is strictly less than the expected number
    LESS_THAN
}

```

Examples:

```

JPPFConnectionPool pool = ...;
// wait until more than 2 connections in the pool are active
List<JPPFClientConnection> list = pool.awaitActiveConnections(Operator.MORE_THAN, 2);
// with the same condition defined as a lambda expression
list = pool.awaitActiveConnections((actual, expected) -> actual > expected, 2);
// wait for no more than 5 seconds or until at least one connection is closed
list = pool.awaitConnections(Operator.AT_LEAST, 1, 5000L,
    JPPFClientConnectionStatus.CLOSED);

```

4.9.5 Associated JMX connection pool

Each connection pool has an associated pool of JMX connections to the same remote driver.. To access and manipulate this JMX pool, the [AbstractClientConnectionPool](#) class provides the following API:

```

public abstract class AbstractClientConnectionPool
    extends AbstractConnectionPool<JPPFClientConnection>
    implements Comparable<AbstractClientConnectionPool> {
    // Get a <i>connected</i> JMX connection among those in the JMX pool
    public JMXDriverConnectionWrapper getJmxConnection()
    // Get a JMX connection among those in the JMX pool
    public JMXDriverConnectionWrapper getJmxConnection(boolean connectedOnly)
    // Get the current maximum size of the associated JMX connection pool
    public int getJMXPoolSize()
    // Set a new size for the associated pool of JMX connections,
    // adding new or closing existing connections as needed
    public int setJMXPoolSize(int maxSize)
}

```

As we can see, the JMX pool handles connections of type [JMXDriverConnectionWrapper](#), which is a wrapper around an [MBeanServerConnection](#), connected to the MBean server of a remote JPPF driver.

Please note that the JMX pool size, when left unspecified, defaults to 1.

[JPPFConnectionPool](#) has higher-level methods to explore the JMX connections with filtering, and wait for them to be in a specified state:

```

public class JPPFConnectionPool extends AbstractClientConnectionPool {
    // Wait for the specified number of JMX connections to be in the specified state
    public List<JMXDriverConnectionWrapper> awaitJMXConnections(
        ComparisonOperator operator, int nbConnections, boolean connectedOnly)
    // Wait for the specified number of JMX connections to be in the specified state,
    // or the specified timeout to expire, whichever happens first
    public List<JMXDriverConnectionWrapper> awaitJMXConnections(
        ComparisonOperator operator, int nbConnections, long timeout, boolean connectedOnly)
    // Wait for a JMX connection to be in the specified state
    public JMXDriverConnectionWrapper awaitJMXConnection(boolean connectedOnly)
}

```


4.9.6 Exploring the connection pools

The [JPPFClient](#) class, or more exactly its super-super class [AbstractJPPFClient](#), provides a number of methods to discover and explore the connection pools currently handled by the client:

```
public class JPPFClient extends AbstractGenericClient { ... }
public abstract class AbstractGenericClient extends AbstractJPPFClient { ... }

public abstract class AbstractJPPFClient
implements ClientConnectionStatusListener, AutoCloseable {
    // Find the connection pool with the specified priority and id
    public JPPFConnectionPool findConnectionPool(int priority, int poolId)
    // Find the connection pool with the specified id
    public JPPFConnectionPool findConnectionPool(int poolId)
    // Find the connection pool with the specified name
    public JPPFConnectionPool findConnectionPool(String name)
    // Find the connection pools whose name matches the specified regular expression
    public List<JPPFConnectionPool> findConnectionPools(String name)
    // Find the connection pools that have at least one connection matching
    // one of the specified statuses
    public List<JPPFConnectionPool> findConnectionPools(
        JPPFClientConnectionStatus...statuses)
    // Get a set of existing connection pools with the specified priority
    public List<JPPFConnectionPool> getConnectionPools(int priority)
    // Get a list of all priorities for the currently existing pools in descending order
    public List<Integer> getPoolPriorities()
    // Get a list of existing connection pools, ordered by descending priority
    public List<JPPFConnectionPool> getConnectionPools()
    // Get a pool with the highest possible priority that has at least 1 active connection
    public JPPFConnectionPool getConnectionPool()
    // Get the connection pools that pass the specified filter
    public List<JPPFConnectionPool> findConnectionPools(
        ConnectionPoolFilter<JPPFConnectionPool> filter)
}
```

Note that the connection pools are held in a multimap-like data structure, where the key is the pool priority sorted in descending order (highest priority first). Consequently, all `getXXX()` and `findXXX()` methods which return a list of connection pools are guaranteed to have the resulting elements of the list sorted by descending priority.

The last `findConnectionPools()` method provides a generic way of filtering the existing connection pools, by making use of a [ConnectionPoolFilter](#), defined as follows:

```
public interface ConnectionPoolFilter<E extends ConnectionPool> {
    // Determine whether this filter accepts the specified connection pool
    boolean accepts(E pool);
}
```

In addition to this, [JPPFClient](#) provides a set of methods which wait until one or more connection pools fulfill a specified condition, and return a list of pools which satisfy the condition:

```
public class JPPFClient extends AbstractGenericClient
// Wait for at least one connection pool with at least one connection in ACTIVE status
public JPPFConnectionPool awaitActiveConnectionPool()
// Wait for at least one connection pool with at least one connection
// in ACTIVE or EXECUTING status
public JPPFConnectionPool awaitWorkingConnectionPool()
// Wait for at least one connection pool with at least one connection
// in one of the specified statuses
public JPPFConnectionPool awaitConnectionPool(JPPFClientConnectionStatus...statuses)
// Wait for at least one connection pool with at least one connection in one of the
// specified statuses or the specified timeout (in ms) expires, whichever happens first
public JPPFConnectionPool awaitConnectionPool(
    long timeout, JPPFClientConnectionStatus...statuses)
// Wait for at least one connection pool with at least one connection in one of the
// specified statuses or the specified timeout (in ms) expires, whichever happens first
public List<JPPFConnectionPool> awaitConnectionPools(
    long timeout, JPPFClientConnectionStatus...statuses)
}
```

4.9.7 Connection Pool Events

The JPPF client API allows the registration or unregistration of listeners to connection pool events: connection pools added to or removed from the client, or connections added to or removed from a connection pool. This can be done in two ways:

1) From a [JPPFClient](#) constructor:

```
public class JPPFClient extends AbstractGenericClient {  
  
    // Initialize this client with an automatically generated application UUID  
    public JPPFClient(final ConnectionPoolListener... listeners)  
  
    // Initialize this client with the specified UUID and listeners  
    public JPPFClient(final String uuid, final ConnectionPoolListener... listeners) {  
    }  
}
```

2) Using the related add and remove methods in the grand parent of JPPFClient: [AbstractJPPFClient](#):

```
public abstract class AbstractJPPFClient  
    implements ClientConnectionStatusListener, AutoCloseable {  
  
    // Add a listener to the list of listeners to this client  
    public void addConnectionPoolListener(final ConnectionPoolListener listener)  
  
    // Remove a listener from the list of listeners to this client  
    public void removeConnectionPoolListener(final ConnectionPoolListener listener)  
}
```

As we can see in the methods signatures, the listeners implement the interface [ConnectionPoolListener](#), is defined as:

```
public interface ConnectionPoolListener extends EventListener {  
  
    // Called when a new connection pool is created  
    void connectionPoolAdded(ConnectionPoolEvent event);  
  
    // Called when a connection pool removed  
    void connectionPoolRemoved(ConnectionPoolEvent event);  
  
    // Called when a new connection is created  
    void connectionAdded(ConnectionPoolEvent event);  
  
    // Called when a connection pool is removed  
    void connectionRemoved(ConnectionPoolEvent event);  
}
```

Note that, if you do not wish to implement all the methods in [ConnectionPoolListener](#), you can instead extend the adapter class [ConnectionPoolListenerAdapter](#), which implements each method of the interface as an empty method.

All notification methods receive an event of type [ConnectionPoolEvent](#), defined as follows:

```
public class ConnectionPoolEvent extends EventObject {  
  
    // Get the source of this event  
    public JPPFConnectionPool getConnectionPool()  
  
    // Get the connection that triggered this event, if any  
    public JPPFClientConnection getConnection()  
}
```

Please note that, in the case of a `connectionPoolAdded()` or `connectionPoolRemoved()` notification, the method `getConnection()` will return `null`, since no connection is involved.

4.9.8 Putting it all together

We will illustrate how client, connection pool events and connection events fit together, with an example which prints the status of all connections created by the client:

```
// this status listener prints the old and new status of each connection
ClientConnectionStatusListener myStatusListener = event -> {
    // obtain the client connection from the event
    JPPFClientConnection connection =
        (JPPFClientConnection) event.getClientConnectionStatusHandler();
    // get the new and old status
    JPPFClientConnectionStatus newStatus = connection.getStatus();
    JPPFClientConnectionStatus oldStatus = event.getOldStatus();
    // print the connection name, old status and new status
    System.out.println(connection.getName() + " status changed from " +
        oldStatus + " to " + newStatus);
};

// create a connection pool listener which registers
// a status listener on new connections
ConnectionPoolListener myPoolListener = new ConnectionPoolListenerAdapter() {
    @Override
    public void connectionAdded(final ConnectionPoolEvent event) {
        // obtain the connection pool and client connection from the event
        JPPFConnectionPool pool = event.getConnectionPool();
        JPPFClientConnection connection = event.getConnection();
        System.out.println("connection " + connection + " added to pool " + pool);
        // add the status listener to the connection
        connection.addClientConnectionStatusListener(myListener);
    }

    @Override
    public void connectionRemoved(final ConnectionPoolEvent event) {
        // obtain the connection pool and client connection from the event
        JPPFConnectionPool pool = event.getConnectionPool();
        JPPFClientConnection connection = event.getConnection();
        System.out.println("connection " + connection + " removed from pool " + pool);
        // remove the status listener from the connection
        connection.removeClientConnectionStatusListener(myListener);
    }
};

// create a JPPFClient with our connection pool listener
JPPFClient client = new JPPFClient(myPoolListener);
```

4.10 Notifications of client job queue events

The JPPF client allows receiving notifications of when jobs are added to or removed from its queue. To this effect, the [AbstractGenericClient](#) class (the super class of [JPPFClient](#)) provides methods to register or unregister listeners for these notifications:

```
public abstract class AbstractGenericClient extends AbstractJPPFClient {
    // Register the specified listener to receive client queue event notifications
    public void addClientQueueListener(ClientQueueListener listener)

    // Unregister the specified listener
    public void removeClientQueueListener(ClientQueueListener listener)
}
```

As we can see, these methods accept listeners of type [ClientQueueListener](#), defined as follows:

```
public interface ClientQueueListener extends EventListener {
    // Called to notify that a job was added to the queue
    void jobAdded(ClientQueueEvent event);

    // Called to notify that a job was removed from the queue
    void jobRemoved(ClientQueueEvent event);
}
```

The `jobAdded()` and `jobRemoved()` methods are notifications of events of type [ClientQueueEvent](#):

```
public class ClientQueueEvent extends EventObject {
    // Get the JPPF client source of this event
    public JPPFClient getClient()

    // Get the job that was added or removed
    public JPPFJob getJob()

    // Get all the jobs currently in the queue
    public List<JPPFJob> getQueuedJobs()

    // Get the size of this job queue
    public int getQueueSize()
}
```

Here is an example usage, which adapts the size of a client connection pool based on the number of jobs in the queue:

```
JPPFClient client = new JPPFClient();
JPPFConnectionPool pool;
// wait until "myPool" is initialized
while ((pool = client.findConnectionPool("myPool")) == null) Thread.sleep(20L);
final JPPFConnectionPool thePool = pool;
// register a queue listener that will adapt the pool size
client.addClientQueueListener(new ClientQueueListener() {

    @Override public void jobAdded(ClientQueueEvent event) {
        int n = event.getQueueSize();
        // grow the connection pool
        JPPFConnectionPool pool = event.getClient().findConnectionPool("myPool");
        if (n > pool.getMaxSize()) pool.setMaxSize(n);
    }

    @Override public void jobRemoved(ClientQueueEvent event) {
        int n = event.getQueueSize();
        // shrink the connection pool
        JPPFConnectionPool pool = event.getClient().findConnectionPool("myPool");
        if (n < pool.getMaxSize()) pool.setMaxSize(n);
    }
});
// ... submit jobs ...
```

4.11 Submitting multiple jobs concurrently

In this section, we will present a number of ways to design an application, such that it can execute multiple jobs concurrently, using a single JPPFClient instance. These can be seen as common reusable patterns, in an attempt at covering the most frequent use cases where submission and processing of multiple jobs in parallel is needed.

The patterns presented here all make the assumption that job submissions are performed through a single instance of [JPPFClient](#). It is the recommended way to work with JPPF, as it benefits the most from the built-in features of the client:

- thread safety
- ability to connect to multiple remote drivers, to the same driver multiple times, or any combination of these
- load-balancing between available connections
- ability to submit a job [over multiple connections](#) for increased performance
- fine-grained filtering of eligible connections for each job, via the job's client-side [execution policy](#)
- connection failover strategies defined via the connection pools priorities

Important note: starting from *JPPF 6.1*, it is no longer required to have multiple connections to the JPPF server to enable concurrent job processing. Each connection can now handle an unlimited number of jobs concurrently.

4.11.1 Job submissions from multiple threads

This pattern explores how concurrent jobs can be submitted by the same [JPPFClient](#) instance by multiple threads. In this pattern, we are using [blocking jobs](#), since each job is submitted in its own thread, thus we can afford blocking that thread until the job completes:

```
public void multipleThreadsBlockingJobs() {
    // a pool of threads that will submit the jobs and retrieve their results
    ExecutorService executor = Executors.newFixedThreadPool(4);
    try (JPPFClient client = new JPPFClient()) {
        // handles for later retrieval of the job submissions results
        List<Future<List<Task<?>>>> futures = new ArrayList<>();
        for (int i=0; i<4; i++) {
            JPPFJob job = new JPPFJob();
            // ... set attributes and add tasks ...
            // submit the job as a Callable in a separate thread
            futures.add(executor.submit(() -> client.submit(job)));
        }
        futures.forEach(future -> {
            try {
                // wait until each job has completed and retrieve its results
                List<Task<?>> results = future.get();
                // ... process the job results ...
                processResults(results);
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }
    executor.shutdown();
}
```

4.11.2 Multiple non-blocking jobs from a single thread

Here, we take advantage of the asynchronous nature of non-blocking jobs to write a much less cumbersome version of the previous pattern:

```
public void singleThreadNonBlockingJobs() {
    try (JPPFClient client = new JPPFClient()) {
        // holds the submitted jobs for later retrieval of their results
        List<JPPFJob> jobs = new ArrayList<>();
        // submit the jobs without blocking the current thread
        for (int i=0; i<4; i++) {
            JPPFJob job = new JPPFJob();
            // ... set other attributes and add tasks ...
            jobs.add(job);
            client.submitAsync(job); // non-blocking operation
        }
        // get and process the jobs results
        jobs.forEach(job -> {
            // synchronize on each job's completion: this is a blocking operation
            List<Task<?>> results = job.awaitResults();
            processResults(results); // process the job results
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

4.11.3 Fully asynchronous processing

Here, we use a [JobListener](#) to retrieve and process the results of the jobs via jobs life cycle notifications. The only synchronization occurs in the main method, to await on the global completion of *all* jobs:

```
public void asynchronousNonBlockingJobs() {
    try (JPPFClient client = new JPPFClient()) {
        int nbJobs = 4;
        // synchronization helper that tells us when all jobs have completed
        final CountDownLatch countDown = new CountDownLatch(nbJobs);
        for (int i=0; i<nbJobs; i++) {
            JPPFJob job = new JPPFJob();
            job.setBlocking(false);
            // results will be processed asynchronously in
            // the job listener's jobEnded() notifications
            job.addJobListener(new JobListenerAdapter() {
                @Override
                public void jobEnded(JobEvent event) {
                    List<Task<?>> results = event.getJob().getAllResults();
                    processResults(results); // process the job results
                    // decrease the jobs count down; when it reaches 0, countDown.await() will exit
                    countDown.countDown();
                }
            });
            // ... set other attributes, add tasks, submit the job ...
            client.submitAsync(job);
        }
        // wait until all jobs are complete, i.e. until the count down reaches 0
        countDown.await();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

4.11.4 Job streaming

Job streaming occurs when an application is continuously creating and executing jobs, based on a potentially infinite source of data. The main problem to overcome in this use case is when jobs are created much faster than they are executed, thus potentially filling the memory until an `OutOfMemoryError` occurs. A possible solution to this is to build a job provider with a limiting factor, which determines the maximum number of jobs that can be running at any given time.

Additionally, an `Iterator` is a Java data structure that fits particularly well the streaming pattern, thus our job provider will implement the `Iterable` interface:

```
public class JobProvider extends JobListenerAdapter
    implements Iterable<JPPFJob>, Iterator<JPPFJob> {
    private int concurrencyLimit; // limit to the maximum number of concurrent jobs
    private int currentNbJobs = 0; // current count of concurrent jobs

    public JobProvider(int concurrencyLimit) {
        this.concurrencyLimit = concurrencyLimit;
    }

    // implementation of Iterator<JPPFJob>
    @Override public synchronized boolean hasNext() {
        boolean hasMoreJobs = false;
        // ... compute hasMoreJobs, e.g. check if there is any more data to read
        return hasMoreJobs;
    }

    @Override public synchronized JPPFJob next() {
        // wait until the number of running jobs is less than the concurrency limit
        while (currentNbJobs >= concurrencyLimit) {
            try {
                wait();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return buildJob();
    }

    @Override public void remove() {
        throw new UnsupportedOperationException("remove() is not supported");
    }

    private synchronized JPPFJob buildJob() {
        JPPFJob job = new JPPFJob();
        // ... build the tasks by reading data from a file, a database, etc...
        // ... add the tasks to the job ...
        job.setBlocking(false);
        // add a listener to update the concurrent jobs count when the job ends
        job.addJobListener(this);
        // increase the count of concurrently running jobs
        currentNbJobs++;
        return job;
    }

    // implementation of JobListener
    @Override synchronized public void jobEnded(JobEvent event) {
        processResults(event.getJob().getAllResults()); // process the job results
        // decrease the count of concurrently running jobs
        currentNbJobs--;
        // wake up the threads waiting in next()
        notifyAll();
    }

    // implementation of Iterable<JPPFJob>
    @Override public Iterator<JPPFJob> iterator() { return this; }

    protected void processResults(List<Task<?>> results) { // ... }
}
```

Note the use of a `JobListener` to ensure the current count of jobs is properly updated, so that the provider can create new jobs from its data source. It is also used to process the job results asynchronously.

Now that we have a job provider, we can use it to submit the jobs it creates to a JPPF grid:


```

public void jobStreaming() {
    try (JPPFClient client = new JPPFClient()) {
        // create the job provider with a limiting concurrency factor
        JobProvider jobProvider = new JobProvider(4);
        // build and submit the provided jobs until no more is available
        jobProvider.forEach(client::submitAsync);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

4.11.4.1 The AbstractJPPFJobStream helper class

Given the potential complexity of the job streaming pattern, we found it useful to provide a helper class which alleviates the work of a developer by implementing all the wiring and internal state transitions, such that the developers can solely focus on the specifics of the jobs they want to submit. The abstract class [AbstractJPPFJobStream](#) serves this purpose. It is defined as follows:

```

public abstract class AbstractJPPFJobStream extends JobListenerAdapter
    implements Iterable<JPPFJob>, Iterator<JPPFJob>, AutoCloseable {

    // Initialize this job provider with a concurrency limit
    public AbstractJPPFJobStream(final int concurrencyLimit) {

        // Determine whether there is at least one more job in the stream
        // This method must be overridden in subclasses
        public abstract boolean hasNext()
        // Get the next job in the stream
        public synchronized JPPFJob next() throws NoSuchElementException
        // Create the next job in the stream, along with its tasks
        // This method must be overridden in subclasses and is called from next()
        protected abstract JPPFJob createNextJob()
        // This operation is not supported
        public void remove() throws UnsupportedOperationException

        // Update the state of this job stream and process the results of a job asynchronously
        public void jobEnded(final JobEvent event)
        // Callback invoked from jobEnded() when a job is complete
        // This method must be overridden in subclasses
        protected abstract void processResults(JPPFJob job)

        // implementation of Iterable<JPPFJob>
        public Iterator<JPPFJob> iterator()

        // Close this stream and release the underlying resources it uses
        // This method must be overridden in subclasses
        public abstract void close() throws Exception

        // Determine whether any job is still being executed
        public boolean hasPendingJob()
        // Get the number of executed jobs
        public int getJobCount()
        // Get the number of executed tasks
        public int getTaskCount()
        // Wait until the end of the job stream has been reached
        public boolean awaitEndOfStream()
    }
}

```

This class is designed to be subclassed and to this effect, we have outlined the four abstract methods that must be overridden in any subclass. We can see how this will simplify the work of any implementation. Let's re-implement the previous example by subclassing [AbstractJPPFJobStream](#):

```

public class JobProvider extends AbstractJPPFJobStream {
    public JobProvider(int concurrencyLimit) {
        super(concurrencyLimit);
    }

    @Override public synchronized boolean hasNext() {
        boolean hasMoreJobs = false;
        // ... compute hasMoreJobs, e.g. check if there is any more data to read
        return hasMoreJobs;
    }

    @Override
    protected JPPFJob createNextJob() {
        JPPFJob job = new JPPFJob();
        // ... build the tasks by reading data from a file, a database, etc...
        // ... add the tasks to the job and return it ...
        return job;
    }

    @Override
    protected void processResults(List<Task<?>> results) {
        // ...
    }

    @Override
    public void close() throws Exception {
        // close a file, database connection, etc...
    }
}

```

4.11.5 Dedicated sample

For a fully working and documented example of the patterns seen in the previous sections, you are invited to explore the dedicated [Concurrent Jobs demo](#).

4.12 Jobs persistence in the driver

As of JPPF 6.0, drivers can persist the jobs they receive, along with their results, in a permanent store. This adds major new capabilities to the drivers, in particular:

- automatic recovery of the jobs in case of driver crashes and resubmission to completion after driver restart, without any external intervention
- the ability to submit jobs from a JPPF client, then check their completion and retrieve their results from a separate client
- the ability to retrieve jobs on demand from a persistent store and resubmit them to completion

To achieve this, JPPF provides the following components:

- a pluggable persistence facility in the driver, with ready to use, built-in implementations
- extensions to the job SLA specifying if and how the persistence of each job should be handled
- a client-side facility to manage, retrieve and monitor persisted jobs

These items are detailed in the next sections.

The persistence facility relies on an implementation of the [JobPersistence](#) interface, whose role is to store, load, delete and query the job elements that are persisted. A job element can be one of 4 types:

- a job header element, which includes the job's uuid, name, [SLA](#), and [metadata](#), but also the information required for job routing and scheduling
- a [data provider](#) element, if any is present
- an element for each task, before its execution
- an element for each executed task returned by the nodes, also known as task execution result

Note: the types of job elements are represented by the [PersistenceObjectType](#) enum.

Each job element is stored as binary data which represents a serialized object graph. Where and how it is stored depends solely on the [JobPersistence](#) implementation. It can be on a file system, a relational database, cloud storage facility, etc.

4.12.1 Job persistence specification

The persistence of a job is specified via a [PersistenceSpec](#) object, defined as follows:

```
public class PersistenceSpec implements Serializable {
    // Determine whether the job is persisted in the driver. Defaults to false
    public boolean isPersistent()
    // Specify whether the job is persisted in the driver
    public PersistenceSpec setPersistent(boolean persistent)

    // Determine whether the driver should automatically execute the persisted job
    // after a restart. Defaults to false
    public boolean isAutoExecuteOnRestart()
    // Specify whether the driver should automatically execute the job after a restart
    public PersistenceSpec setAutoExecuteOnRestart(boolean autoExecuteOnRestart)

    // Determine whether the persisted job should be deleted from the store upon
    // completion. Defaults to true
    public boolean isDeleteOnCompletion()
    // Determine whether the job should be deleted from the store upon completion
    public PersistenceSpec setDeleteOnCompletion(boolean deleteOnCompletion)
}
```

As we can see, instances of this class manage three boolean flags which specify whether the driver will persist the job and what it will do with a persisted job when it completes or when a driver restart occurs.

The **"persistent"** flag determines whether the job is persisted at all. By default, it is set to **false**, which means that a job is not persisted by default. When set to true, the driver will persist the job elements it receives from the client, and later on the execution results received from the nodes. When a job is not configured as persistent, it will be processed without any overhead. If this flag is set to false, none of the other flags has any effect.

The **"delete on completion"** flag determines whether the job should be removed from the store when it completes. This addresses situations where a client remains connected to the driver and awaits the job results for further processing, while you still want the driver to be able to recover the job in case of a crash followed by a restart. Since the client receives the results, they no longer need to be kept in the permanent store. This flag is set to **true** by default.

The **"auto execute on restart"** flag tells a driver that, upon restart, it should automatically resubmit the job's unexecuted tasks until the job completes. At startup, the driver will retrieve all the jobs with this flag set to true that have not yet completed, and resubmit them. This flag is set to **false** by default.

Note: when the **"auto execute on restart"** flag is true, the nodes to which unexecuted tasks are dispatched still need access to all the required classes for these tasks. The easiest way to achieve this is to add the corresponding jar files and class folders to the driver's classpath and let the nodes' distributed class loader find them there.

As an example, here is how we would configure a persistent job that should be automatically executed upon driver restart and deleted from the store upon completion:

```
JPPFJob job = new JPPFJob();
job.getSLA().getPersistenceSpec()
    .setPersistent(true)
    .setAutoExecuteOnRestart(true)
    .setDeleteOnCompletion(true);
```

Note: if your intended usage scenario is to submit a job, close the client application, then query the job results later on, you **MUST** set the job's [cancelUponClientDisconnect](#) flag to false. This will prevent the job from being cancelled when disconnecting the client.

4.12.2 Managing persisted jobs

In order to manage and query the jobs in a persistence store, JPPF provides a client-side facility, based on JMX management APIs, which connects to a remote driver and accesses its persistence store. This facility is implemented in the [JPPFDriverJobPersistence](#) class, defined as follows:

```
public class JPPFDriverJobPersistence {
    // Initialize this persisted job manager with the specified driver JMX connection
    public JPPFDriverJobPersistence(JMXDriverConnectionWrapper jmx)

    // List the persisted jobs that match the provided job selector
    public List<String> listJobs(JobSelector selector) throws Exception

    // Delete the persisted job with the specified uuid.
    // This method is equivalent to deleteJobs(new JobUuidSelector(uuid))
    public boolean deleteJob(String uuid) throws Exception

    // Delete the persisted jobs that match the provided job selector
    public List<String> deleteJobs(JobSelector selector) throws Exception

    // Retrieve and rebuild the persisted job with the specified uuid
    // This method is equivalent to retrieveJob(uuid, false)
    public JPPFJob retrieveJob(String uuid) throws Exception

    // Get the description of the job with the specified uuid.
    // This method retrieves the job's uuid, name, number of tasks, SLA and metadata
    public JPPFDistributedJob getJobDescription(String uuid) throws Exception

    // Determines whether the job has completed and all execution results are available
    public boolean isJobComplete(String uuid) throws Exception
}
```

Note that the constructor for this class takes a [JMXDriverConnectionWrapper](#), so that it can establish a JMX connection to the remote driver. A [JMXDriverConnectionWrapper](#) can be obtained in 2 ways:

- by creating it directly, for example:

```
JMXDriverConnectionWrapper jmx =
    new JMXDriverConnectionWrapper("my.driver.com", 11198, false);
jmx.connectAndWait(3000L);
JPPFDriverJobPersistence jobPersistence = new JPPFDriverJobPersistence(jmx);
```

- or by getting it from a [JPPFClient](#), for example:

```
JPPFClient client = new JPPFClient();
JMXDriverConnectionWrapper jmx =
    client.awaitWorkingConnectionPool().awaitWorkingJMXConnection();
JPPFDriverJobPersistence jobPersistence = new JPPFDriverJobPersistence(jmx);
```

4.12.2.1 Listing the persisted jobs

This operation can be achieved with the `listJobs()` method of [JPPFDriverJobPersistence](#). This method takes a [job selector](#) and returns a list of the uuids of the persisted jobs that match the selector. If no persisted job matches, then the returned list is empty. The uuids in the list can then be reused in other methods of [JPPFDriverJobPersistence](#).

Example usage:

```
JPPFClient client = new JPPFClient();
// create the job persistence manager instance
JPPFDriverJobPersistence jobPersistence = new JPPFDriverJobPersistence(
    client.awaitWorkingConnectionPool().awaitWorkingJMXConnection());

// list the uuids of all currently persisted jobs
List<String> uuids = jobPersistence.listJobs(JobSelector.ALL_JOBS);
for (String uuid: uuids) {
    // do something with each job uuid
}
```

4.12.2.2 Retrieving a job

To retrieve a job from a persistence store, you use the `retrieveJob()` method of [JPPFDriverJobPersistence](#). This method takes a job uuid and returns a [JPPFJob](#) that can be used as if it had been created directly on the client side. For instance, you can resubmit it if it still has unexecuted tasks, or process its results if it has completed.

Here is an example usage:

```
JPPFClient client = new JPPFClient();
JPPFDriverJobPersistence jobPersistence = ...;

// list the uuids of all currently persisted jobs
List<String> uuids = jobPersistence.listJobs(JobSelector.ALL_JOBS);

// retrieve all the persisted jobs and resubmit those that haven't completed yet
LinkedList<JPPFJob> jobs = new LinkedList<>();
for (String uuid: uuids) {
    JPPFJob job = jobPersistence.retrieveJob(uuid);
    jobs.add(job);
    // delete the job from the store, it will be stored again if resubmitted
    jobPersistence.deleteJob(uuid);
    // if the job has unexecuted tasks, resubmit it
    if (job.unexecutedTaskCount() > 0) {
        jobs.addLast(job);
        client.submitAsync(job);
    } else {
        jobs.addFirst(job);
    }
}

// process the results of all jobs
for (JPPFJob job: jobs) {
    // if the job has already completed, this method returns immediately
    List<Task<?>> results = job.awaitResults();
    // ... process the results ...
}
```

4.12.2.3 Deleting one or more jobs

To delete one or more jobs from the persistence store, you can use either:

- the `deleteJob(String)` method to delete a single job at a time. This method takes a job uuid and returns `true` if a job with this uuid was found and effectively deleted, `false` otherwise.
- or the `deleteJobs(JobSelector)` method. This method takes a [JobSelector](#) and returns a list of the uuids of the jobs that were found and effectively deleted.

The following example deletes all persisted jobs using `deleteJob(String)` in a loop:

```
JPPFDriverJobPersistence jobPersistence = ...;
// list all currently persisted jobs
List<String> uuids = jobPersistence.listJobs(JobSelector.ALL_JOBS);
// delete all the persisted jobs
for (String uuid: uuids) {
    // delete the job from the store and check success
    if (jobPersistence.deleteJob(uuid)) {
        System.out.println("sucessfully deleted job with uuid = " + uuid);
    }
}
```

This example performs exactly the same thing using `deleteJobs(JobSelector)` without a loop:

```
JPPFDriverJobPersistence jobPersistence = ...;
// delete all persisted jobs
List<String> deletedUuids = jobPersistence.deleteJobs(JobSelector.ALL_JOBS);
System.out.println("sucessfully deleted jobs with uuids = " + deletedUuids);
```

4.12.2.4 Getting information on the jobs

[JPPFDriverJobPersistence](#) provides 2 methods to obtain information on a persisted job:

- `isJobComlete()` determines whether a job has completed its execution. It takes a job uuid as input and returns `true` if the job has completed, `false` otherwise
- `getJobDescription()` provides detailed information on the job name, number of tasks, SLA and metadata. It takes a job uuid and returns an instance of [JPPFDistributedJob](#) which encapsulates the details.

The following example illustrates a possible usage of these methods:

```
JPPFDriverJobPersistence jobPersistence = ...;

// retrieve all the jobs submitted by "john.doe"
String script = "'john.doe'.equals(jppfJob.getMetadata().getParameter('submitter'))";
JobSelector selector = new ScriptedJobSelector("javascript", script);
List<String> uuids = jobPersistence.listJobs(selector);
for (String uuid: uuids) {
    if (jobPersistence.isJobComplete(uuid)) {
        // ... process completed job ...
    } else {
        JPPFDistributedJob jobDesc = jobPersistence.getJobDescription(uuid);
        // ... do something with the job description ...
    }
}
```

4.12.3 Configuring jobs persistence in the driver

A job persistence facility is essentially a pluggable implementation of the [JobPersistence](#) interface. A driver supports a single persistence implementation at a time, configured through the `"jppf.job.persistence"` configuration property:

```
jppf.job.persistence = <implementation class name> param1 ... paramN
```

where:

- **"implementation class name"** is the fully qualified class name of the [JobPersistence](#) implementation
- **"param1 ... paramN"** are optional string parameters used by the persistence implementation

For example, to configure the [default file persistence](#) with a root directory named "persistence" in the driver's working directory, we would configure the following:

```
jppf.job.persistence = org.jppf.job.persistence.impl.DefaultFilePersistence persistence
```

If no job persistence is configured, the driver will default to the default file persistence with a root directory "persistence", exactly as in the example above.

4.12.4 Built-in persistence implementations

4.12.4.1 Default file persistence

The default file persistence is a file-based persistent store for jobs. The corresponding implementation class is [DefaultFilePersistence](#).

The store's structure is made of a root directory, under which there is one directory per job, named after the job's uuid. Each job directory contains:

- a file named "header.data" for the job header
- a file named "data_provider.data" for the job's data_provider
- a file named "task-i.data" for each task *i* of the job, where *i* represents the position of the task in the job
- a file "result-i.data" for each task result *i* received from a node, where *i* represents the position of the task in the job

For example, if we define the file persistence with a root directory named "persistence" in the driver's working directory:

```
jppf.job.persistence = org.jppf.job.persistence.impl.DefaultFilePersistence persistence
```

Let's say we submitted a job with two tasks which ran to completion, with a uuid of "my_job_uuid". The persistence store structure would then look like this:

```
JPPF-6.0-driver
|_persistence
  |_my_job_uuid
    |_header.data
    |_data_provider.data
    |_task-0.data
    |_task-1.data
    |_result-0.data
    |_result-1.data
```

Important note: when a job element (header, data provider, task or result) is stored, its data is first put into a temporary file with the ".tmp" extension. When the data is fully stored in the file, and only then, the file is renamed with a ".data" extension. This avoids ending up with incomplete or corrupted files that would prevent JPPF from restoring the job.

The built-in file persistence is configured as:

```
jppf.job.persistence = org.jppf.job.persistence.impl.DefaultFilePersistence <root_dir>
```

where *root_dir* can be either an absolute file path, or a path relative to the driver's working directory. If it is omitted, it defaults to "persistence".

Tip: you may also use [system properties substitutions](#) in your configuration to specify common paths. For instance, to point the file persistence to a root directory named "jppf_jobs" in the current user's home directory, we could write:

```
jppf.job.persistence = org.jppf.job.persistence.impl.DefaultFilePersistence \
  ${sys.user.home}/jppf_jobs
```

4.12.4.2 Default database persistence

The default database persistence is a job persistence implementation which stores jobs in a single database table. Its corresponding implementation class is [DefaultDatabasePersistence](#), and it is configured as follows:

```
jppf.job.persistence = org.jppf.job.persistence.impl.DefaultDatabasePersistence \
  <table_name> <datasource_name>
```


where:

- "**table_name**" is the name of the table in which jobs are stored
- "**datasource_name**" is the name of a datasource configured via the [database services](#) facility

If both *table_name* and *datasource_name* are omitted, they will default to "JOB_PERSISTENCE" and "job_persistence", respectively. If *datasource_name* is omitted, it will default to "job_persistence".

The following is an example configuration with a table named "JPPF_JOBS" and a datasource named "jobDS":

```
# persistence definition
jppf.job.persistence = org.jppf.job.persistence.impl.DefaultDatabasePersistence \
    JPPF_JOBS jobDS

# datasource definition
jppf.datasource.jobs.name = jobDS
jppf.datasource.jobs.driverClassName = com.mysql.jdbc.Driver
jppf.datasource.jobs.jdbcUrl = jdbc:mysql://localhost:3306/testjppf
jppf.datasource.jobs.username = testjppf
jppf.datasource.jobs.password = testjppf
jppf.datasource.jobs.minimumIdle = 5
jppf.datasource.jobs.maximumPoolSize = 10
jppf.datasource.jobs.connectionTimeout = 30000
jppf.datasource.jobs.idleTimeout = 600000
```

The table structure is as follows:

```
CREATE TABLE <table_name> (
    UUID varchar(250) NOT NULL,
    TYPE varchar(20) NOT NULL,
    POSITION int NOT NULL,
    CONTENT blob NOT NULL,
    PRIMARY KEY (UUID, TYPE, POSITION)
);
```

Where:

- the **UUID** column represents the job uuid
- the **TYPE** column represents the type of object, taken from the [PersistenceObjectType](#) enum
- the **POSITION** column represents the object's position in the job, if TYPE is 'task' or 'task_result', otherwise -1
- the **CONTENT** column represents the serialized job element

Very important: the table definition should be adjusted depending on the database you are using. For instance, in MySQL the BLOB type has a size limit of 64 KB, thus storing job elements larger than this size will always fail. In this use case, the MEDIUMBLOB or LONGBLOB type should be used instead.

If the table does not exist, JPPF will attempt to create it. If this fails for any reason, for instance if the database user does not have sufficient privileges, then persistence will be disabled.

It is possible to specify the path to the file that contains the DDL statement(s) to create the table, like so:

```
# path to the file or resource containing the DDL statements to create the table
jppf.job.persistence.ddl.location = <ddl_path>
```

Here, *<ddl_path>* is the path to either a file in the file system or a resource in the class path. The file system is always looked up first and if no file is found, then JPPF looks up in the driver's classpath. The default value for this property is "[org/jppf/job/persistence/impl/job_persistence.sql](#)", which points to a file in the classpath which can be found in the **jppf-common-x.y.z.jar** file.

4.12.4.3 Asynchronous persistence (write-behind)

Asynchronous persistence is an asynchronous wrapper for any other job persistence implementation. It delegates the persistence operations to this other implementation, and executes the delegated operations asynchronously via a pool of threads. The corresponding implementation class is [AsynchronousPersistence](#).

The methods of [JobPersistence](#) that do not return a result (void return type) are non-blocking and return immediately. All other methods will block until the delegated operation is executed and its result is available. In particular, all operations that store job data are executed some time in the future, which makes this implementation an effective "write-behind" facility.

Asynchronous persistence can be configured as follows:

```
# shorten the configuration value for clarity
wrapper = org.jppf.job.persistence.impl.AsynchronousPersistence
# asynchronous persistence with a specified thread pool size
jppf.job.persistence = ${wrapper} <pool_size> <actual_persistence> <param1> ... <paramN>
```

Where:

- "**pool_size**" is the size of the pool of threads used to execute the delegated operations. It can be omitted, in which case it defaults to 1 (single-threaded).
- "**<actual_persistence> <param1> ... <paramN>**" is the configuration of the delegate persistence implementation

Here is an example configuration for an asynchronous database persistence:

```
pkg = org.jppf.job.persistence.impl
# asynchronous database persistence with pool of 4 threads,
# a table named 'JPPF_JOBS' and datasource named 'JobDS'
jppf.job.persistence = ${pkg}.AsynchronousPersistence 4 \
    ${pkg}.DefaultDatabasePersistence JPPF_JOBS JobDS
```

Performance implications: *the main goal of the asynchronous persistence is to minimize the impact of persistence on performance, at the risk of a greater data loss in case of a driver crash. In scenarios where jobs are submitted and executed faster than they are persisted, they will tend to accumulate in the thread pool's queue, with a risk of an out of memory condition if the excessive load persists for too long.*

To mitigate this possible issue, the asynchronous persistence monitors the heap usage. When heap usage reaches a given threshold, it stops asynchronous operations and delegates directly to the underlying persistence implementation instead, until the heap usage drops back under the threshold.

The heap usage threshold is the ratio `used_heap / max_heap` expressed as a percentage. It has a default value of 70% and can be set with the following configuration property:

```
jppf.job.persistence.memory.threshold = 60.0
```

4.12.4.4 Cacheable persistence

The cacheable persistence is a caching wrapper for any other job persistence implementation, whose corresponding implementation class is [CacheablePersistence](#).

The cached artifacts are those handled by the `load()` and `store()` methods, that is, job headers, data providers, tasks and task results. The cache is an LRU cache of soft references to the artifacts. It guarantees that all its entries will be garbage-collected before an out of memory error is raised. Additionally the cache has a capacity that can be specified in the configuration and which defaults to 1024.

This cacheable persistence is configured as follows:

```
# shorten the configuration value for clarity
wrapper = org.jppf.job.persistence.impl.CacheablePersistence
# cacheable persistence with a specified capacity
jppf.job.persistence = ${wrapper} <capacity> <actual_persistence> <param1> ... <paramN>
```

Where:

- "**capacity**" is the capacity of the cache, that is, the maximum number of entries it can hold at any time. If omitted, it defaults to 1024.
- "**<actual_persistence> <param1> ... <paramN>**" is the configuration of the delegate persistence implementation

Here is a concrete example wrapping a default database persistence:

```
# shortcut for the package name
pkg = org.jppf.job.persistence.impl
# cacheable database persistence with a capacity of 10000,
# a table named 'JPPF_JOBS' and datasource named 'JobDS'
jppf.job.persistence = ${pkg}.CacheablePersistence 10000 \
    ${pkg}.DefaultDatabasePersistence JPPF_JOBS JobDS
```

Note: *since the job persistence facility, by its nature and design, performs mostly "write" operations, you will generally see little or no benefit to using the cacheable persistence wrapper. You may see significant performance gains essentially in situations where the persisted jobs are accessed multiple times by the client-side [management facility](#).*

Tip: it is possible to combine cacheable persistence with asynchronous persistence to wrap any concrete persistence implementation. This is done by simply concatenating the class names and related arguments in the configuration, e.g.:

```
# shortcuts for package name and persistence implementations
pkg = org.jppf.job.persistence.impl
cacheable = ${pkg}.CacheablePersistence 1024
async = ${pkg}.AsynchronousPersistence 8
db = ${pkg}.DefaultDatabasePersistence JPPF_JOBS jobDS

# combine them to configure a cacheable asynchronous database persistence
jppf.job.persistence = ${cacheable} ${async} ${db}
```

4.12.5 Custom persistence implementations

Reference: custom implementations are fully detailed in a [dedicated section](#) of the [customization](#) chapter.

4.12.6 Class loading and classpath considerations

In a scenario where you want a job to be automatically resubmitted by the persistence facility, after a driver restart and without any client connected, the nodes to which the tasks of the job are dispatched will need to be able to deserialize these tasks and then execute them. For this, they will need to load all the classes required by these tasks, otherwise the execution will fail. Normally, these classes would be downloaded from the client via the driver, however that is not possible here, since there is no client.

To ensure that these classes can be loaded, they must be in a place accessible from either the nodes or the driver. Our recommendation is to put the corresponding jar files and class folders **in the driver's classpath**, to ensure all the nodes can access them.

Reference: for details on how class loading works in JPPF, please read the [class loading](#) documentation.

Similarly, when restoring jobs on the client side with the [JPPFDriverJobPersistence](#) facility, you must ensure that all required classes are available in the client classpath, to allow job elements to be deserialized properly.

4.12.7 Jobs persistence in multi-server topologies

In topologies that include multiple drivers, whether they communicate with each other and/or the JPPF client is connected to one or more of them at once, there are scenarios that require special handling and care in order to guarantee the integrity of the jobs in the persistence store.

4.12.7.1 Only the first peer persists the job

Consider a topology with 2 drivers communicating each other. When a JPPF client submits a job to driver 1, and driver 1 delegates all or a part of the job to driver 2, then only driver 1 will persist the job. This simplifies the handling of persistence, avoids redundant persistence operations and generally increases performance.

4.12.7.2 Submitting a job via multiple driver connections:

The client-side load balancer, combined with the job's client SLA [maxChannels](#) attribute, allows JPPF clients to submit jobs via multiple connections in parallel. We distinguish 2 cases with regards to jobs persistence:

- 1) All connections point to the same driver. In this case no special handling is needed, because the same driver also means the same persistence store.
- 2) The connections point to 2 or more separate drivers. In this scenario, two conditions must be met:
 - all the drivers must point to the same persistence store
 - the persistence store must provide some sort of transactionality or locking mechanism to protect against integrity constraint violations. In effect, some elements of the job might be stored multiple times in parallel by multiple drivers and the store must be protected against possible corruption. Relational databases will generally provide the transactional capabilities to achieve this. For instance the [built-in database persistence](#) does, but the [built-in file persistence](#) does not.

4.13 JPPF Executor Services

4.13.1 Basic usage

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

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

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

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

Here is an example use:

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

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

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

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

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

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

4.13.2 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 $\text{current batch size} + n > \text{limit}$. 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();
```

4.13.3 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 `ExecutorService`, `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();
```

4.13.3.1 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 jobs state
    <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);
    // get all the class loaders added to this job configuration
    List<ClassLoader> getClassLoaders();
}
```

As we can see, this provides a way to set the properties normally available to [JPPFJob](#) instances, even though the jobs submitted 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 configuration
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));
// add a class loader that cannot be computed from the tasks
jobConfig.getClassLoaders().add(myClassLoader);
```


4.13.3.2 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");
    }
}
```

4.13.4 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> completionService =
    new JPPFCompletionService<String>(executor);
MyCallable<String> task = new MyCallable<String>();
Future<String> future = completionService.submit(task);

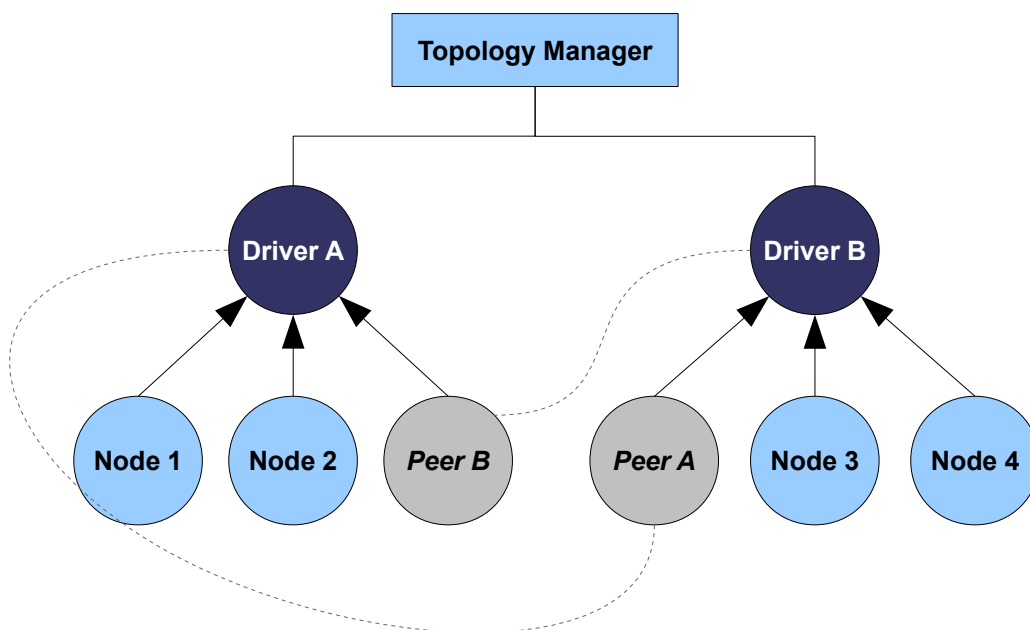
// ... later on ...
// block until a result is available
future = completionService.take();
String result = future.get();
```


4.14 Grid topology monitoring

As of JPPF v5.0, the package `org.jppf.client.monitoring.topology` provides an API to explore and monitor the topology of a JPPF grid, such as discovered from a JPPF client's perspective. This API is used by the administration and monitoring console, and allows to explore the topology as well as subscribe to notifications of the changes that occur, such as new server being brought online, nodes started or terminated, etc.

4.14.1 Building and exploring the topology

The entry point for this API is the [TopologyManager](#) class. A `TopologyManager` uses a [JPPFClient](#) to discover its connections to one or more drivers, and queries these drivers via the management API to obtain information about their attached JPPF nodes and other peer drivers they are connected to. All this information is used to build a model of the JPPF grid topology as a tree. For example, with a grid that has two drivers connected to each other, we would have a representation like this:



The dotted lines emphasize the fact that a peer is actually a "virtual" component which references a real driver. Peer objects are here to allow representing the topology as a tree rather than as a graph, which makes their usage a lot easier.

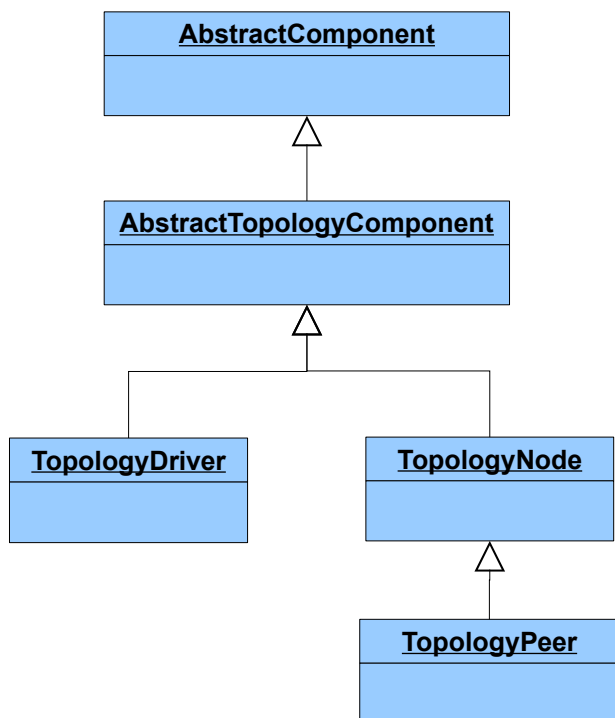
You may notice that this is very similar to the "topology tree" view of the administration console:

Node	Threads ...	Node Status	Exec status	Tasks Executed	Nb slave nodes
192.168.1.24:11191					
192.168.1.24:11192					
192.168.1.24:12001	8 / 5	Connected	Idle	0	1
192.168.1.24:12002	8 / 5	Connected	Idle	0	
192.168.1.24:11192					
127.0.0.1:12003	8 / 5	Connected	Idle	0	0
127.0.0.1:12004	8 / 5	Connected	Idle	0	0
192.168.1.24:11191					

This is not a coincidence: the administration console indeed uses its own instance of [TopologyManager](#) and the tree view is a direct mapping of the topology to a tree-like swing component.

The model representing the components of the JPPF topology is a class hierarchy whose base super class is the abstract class [AbstractTopologyComponent](#), which provides an API to navigate within the tree structure, along with ways to identify each component and methods to retrieve information that are common to drivers and nodes.

The object model is as follows:



[AbstractComponent](#) provides the API to navigate a tree of elements uniquely identified with a uuid:

```
public abstract class AbstractComponent<E extends AbstractComponent> {
    // Get the parent of this component
    public synchronized E getParent() {
    // Get the number of children of this component
    public synchronized int getChildCount()
    // Get the children of this component
    public synchronized List<E> getChildren()
    // Get the child with the specified uuid
    public synchronized E getChild(final String uuid)
    // Get the uuid of this component
    public synchronized String getUuid()
    // Get a user-friendly representation of this topology component
    public String getDisplayName()
}
```

[AbstractTopologyComponent](#) is defined as follows:

```
public abstract class AbstractTopologyComponent
    extends AbstractComponent<AbstractTopologyComponent> {
    // Whether this object represents a driver
    public boolean isDriver()
    // Whether this object represents a node
    public boolean isNode()
    // Whether this object represents a peer driver
    public boolean isPeer()

    // Get the object describing the health of a node or driver
    public HealthSnapshot getHealthSnapshot()
    // Get the management information for this component
    public JPPFManagementInfo getManagementInfo()
    // Get a user-friendly representation of this component
    public String getDisplayName()
}
```

[TopologyDriver](#) provides the following additional methods:

```
public class TopologyDriver extends AbstractTopologyComponent {
    // Get the JMX connection wrapper
    public JMXDriverConnectionWrapper getJmx()
    // Get the driver connection
    public JPPFClientConnection getConnection()
    // Get a proxy to the MBean that forwards node management requests
    public JPPFNodeForwardingMBean getForwarder()
    // Get a proxy to the jobs monitoring MBean
    public DriverJobManagementMBean getJobManager()
    // Get a proxy the diagnostics MBean for this driver
    public DiagnosticsMBean getDiagnostics()
    // Get the nodes attached to this driver as TopologyNode objects
    public List<TopologyNode> getNodes()
    // Get the peers connected to this driver as TopologyPeer objects
    public List<TopologyPeer> getPeers()
    // Get the nodes and peers connected to this driver
    public List<TopologyNode> getNodesAndPeers()
}
```

[TopologyNode](#) provides two specialized method to query the node's state:

```
public class TopologyNode extends AbstractTopologyComponent {
    // Get the object describing the current state of a node
    public JPPFNodeState getNodeState()
    // Get the number of slaves for a master node (node provisioning)
    public int getNbSlaveNodes()
}
```

Whereas [TopologyPeer](#) does not provide any additional method, since its uuid is all that is needed.

The root of the tree is our [TopologyManager](#), defined as follows:

```
public class TopologyManager implements ClientListener {
    // Create a topology manager with a new JPPFClient
    public TopologyManager()
    // Create a topology manager with the specified JPPFClient
    public TopologyManager(JPPFClient client)
    // Get the JPPF client
    public JPPFClient getJPPFClient()
    // Get all drivers
    public List<TopologyDriver> getDrivers()
    // Get all nodes
    public List<TopologyNode> getNodes()
    // Get all peers
    public List<TopologyPeer> getPeers()
    // Get a driver from its uuid
    public TopologyDriver getDriver(String uuid)
    // Get a node from its uuid
    public TopologyNode getNode(String uuid)
    // Get a peer from its uuid
    public TopologyPeer getPeer(String uuid)
    // Get a node or peer from its uuid
    public TopologyNode getNodeOrPeer(String uuid)
    // Get the number of drivers
    public int getDriverCount()
    // Get the number of nodes
    public int getNodeCount()
    // Get the number of peers
    public int getPeerCount()
    // Get and set the node filter
    public NodeSelector getNodeFilter()
    public void setNodeFilter(NodeSelector selector)
    // Get the nodes that are slaves of the specified master node, if any
    public List<TopologyNode> getSlaveNodes(String masterNodeUuid)
}
```

According to this, the following code example can be used to visit the entire tree:

```
// this creates a new JPPFClient accessible with manager.getJPPFClient()
TopologyManager manager = new TopologyManager();
// iterate over the discovered drivers
for (TopologyDriver driver: manager.getDrivers()) {
    // ... do something with the driver ...
    // iterate of the nodes and peers for this driver
    for (TopologyNode node: driver.getNodesAndPeers()) {
        if (comp.isNode()) {
            // ... do something with the node ...
        } else { // if (comp.isPeer())
            TopologyPeer peer = (TopologyPeer) node;
            // retrieve the actual driver the peer refers to
            TopologyDriver actualDriver = manager.getDriver(peer.getUuid());
            // ... do something with the peer ...
        }
    }
}
```

A [TopologyManager](#) instance also automatically refreshes the states of the nodes, along with the JVM health snapshots of the drivers and nodes. The interval between refreshes is determined via the value of the following configuration properties:

```
# refresh interval in millis for the grid topology.
# this is the interval between 2 successive runs of the task that refreshes the
# topology via JMX requests; defaults to 1000
jppf.admin.refresh.interval.topology = 1000

# refresh interval for the JVM health snapshots; defaults to 1000
# this is the interval between 2 successive runs of the task that refreshes
# the JVM health snapshots via JMX requests
jppf.admin.refresh.interval.health = 1000
```

These values can be set both in a configuration file and programmatically, for instance:

```
TypedProperties config = JPPFConfiguration.getProperties();
// refresh the nodes states every 3 seconds
config.setLong("jppf.admin.refresh.interval.topology", 3000L);
// refresh the JVM health snapshots every 5 seconds
config.setLong("jppf.admin.refresh.interval.health", 5000L);
TopologyManager manager = new TopologyManager();
```

4.14.2 Receiving notifications of changes in the topology

It is possible to subscribe to topology change events emitted by a [TopologyManager](#), using an implementation of the [TopologyListener](#) interface as parameter of the related constructors and methods in [TopologyManager](#):

```
public class TopologyManager implements ClientListener {
    // Create a topology manager with a new JPPFClient
    // and add the specified listeners immediately
    public TopologyManager(TopologyListener...listeners)
    // Create a topology manager with the specified JPPFClient
    // and add the specified listeners immediately
    public TopologyManager(JPPFClient client, TopologyListener...listeners)

    // Add a topology change listener
    public void addTopologyListener(TopologyListener listener)
    // Remove a topology change listener
    public void removeTopologyListener(TopologyListener listener)
}
```

[TopologyListener](#) is defined as follows:

```
public interface TopologyListener extends EventListener {
    // Called when a driver is discovered
    void driverAdded(TopologyEvent event);
    // Called when a driver is terminated
    void driverRemoved(TopologyEvent event);
    // Called when the state of a driver has changed
    void driverUpdated(TopologyEvent event);
    // Called when a node is added to a driver
    void nodeAdded(TopologyEvent event);
    // Called when a node is removed from a driver
    void nodeRemoved(TopologyEvent event);
    // Called when the state of a node has changed
    void nodeUpdated(TopologyEvent event);
}
```

As we can see, each notification is encapsulated in a [TopologyEvent](#) object:

```
public class TopologyEvent extends EventObject {
    // Get the driver data
    public TopologyDriver getDriver()
    // Get the related node or peer
    public TopologyNode getNodeOrPeer()
    // Get the topology manager which emitted this event
    public TopologyManager getTopologyManager()
}
```

Please note that, for the `driverAdded(...)`, `driverRemoved(...)` and `driverUpdated(...)` notifications, the corresponding event's `getNodeOrPeer()` method will always return null.

Additionally, if you do not wish to override all the methods of the [TopologyListener](#) interface, you can instead extend the class [TopologyListenerAdapter](#), which provides an empty implementation of each method.

Here is a listener implementation that prints topology changes to the console:

```
public class MyListener extends TopologyListenerAdapter {
    @Override public void driverAdded(TopologyEvent e) {
        System.out.printf("added driver %s\n", e.getDriver().getDisplayName());
    }

    @Override public void driverRemoved(TopologyEvent e) {
        System.out.printf("removed driver %s\n", e.getDriver().getDisplayName());
    }

    @Override public void nodeAdded(TopologyEvent e) {
        TopologyNode node = e.getNodeOrPeer();
        System.out.printf("added %s %s to driver %s\n", nodeType(node),
            node.getDisplayName(), e.getDriver().getDisplayName());
    }

    @Override public void nodeRemoved(TopologyEvent e) {
        TopologyNode node = e.getNodeOrPeer();
        System.out.printf("removed %s %s from driver %s\n", nodeType(node),
            node.getDisplayName(), e.getDriver().getDisplayName());
    }

    private String nodeType(TopologyNode node) {
        return node.isNode() ? "node" : "peer";
    }
}
```

To subscribe this listener for topology changes:

```
TopologyManager manager = new TopologyManager();
manager.addTopologyListener(new MyListener());
```

or, in a single statement:

```
TopologyManager manager = new TopologyManager(new MyListener());
```

4.14.3 Node filtering

[TopologyManager](#) provides an API to filter the nodes in the grid topology, based on [node selectors](#):

```
public class TopologyManager implements ClientListener {
```

```
// Get the node filter
public NodeSelector getNodeFilter()
// Set the node filter
public void setNodeFilter(NodeSelector selector)
}
```

To activate the node filtering, you only need to call `setNodeFilter()` with a non-null [NodeSelector](#). Inversely, calling `setNodeFilter()` with a null selector will deactivate the node filtering. Here is an example which filters out all slave nodes in the topology:

```
TopologyManager manager = ...;
// a policy that filters out slave nodes
ExecutionPolicy noSlavePolicy = new Equal("jppf.node.provisioning.slave", false);
NodeSelector selector = new ExecutionPolicySelector(noSlavePolicy);
// activate node filtering with our policy
manager.setNodeFilter(selector);
```

When node filtering is active:

- only the nodes that match the selector will be available via the TopologyManager API
- events will be emitted only for the nodes that match the selector

The administration console also provides a UI to enter an execution policy in XML format to use as a filter:

The screenshot displays the JPPF Monitoring and Administration Tool interface. The main window shows a table of nodes with columns: Driver / Node, Threads / Pri..., Exec status, Executed T..., Slave nodes, and Pending. The table lists a driver node 'localhost:11191' and two slave nodes 'Icohen-CSL:12001' and 'Icohen-CSL:12002'. Below the main window, a 'View-1' window is open, showing the 'Filtering' configuration. It includes a toolbar with icons for filtering, and a text area containing XML code for an `ExecutionPolicy` of type `Equal` with the property `jppf.node.provisioning.slave` set to `false`. A tooltip 'Activate/deactivate filtering based on this filter' is visible over the filter icon. At the bottom, a status bar shows 'Active servers: 1' and 'Active nodes: 2'.

Driver / Node	Threads / Pri...	Exec status	Executed T...	Slave nodes	Pending
localhost:11191					
Icohen-CSL:12001	8 / 5	Idle	0	4 None	
Icohen-CSL:12002	8 / 5	Idle	0	4 None	

```
<!-- When this filter is active, only the nodes matching the
executi Activate/deactivate filtering based on this filter admin console
-->
<ExecutionPolicy>
  <Equal valueType="boolean">
    <Property>jppf.node.provisioning.slave</Property>
    <Value>>false</Value>
  </Equal>
</ExecutionPolicy>
```

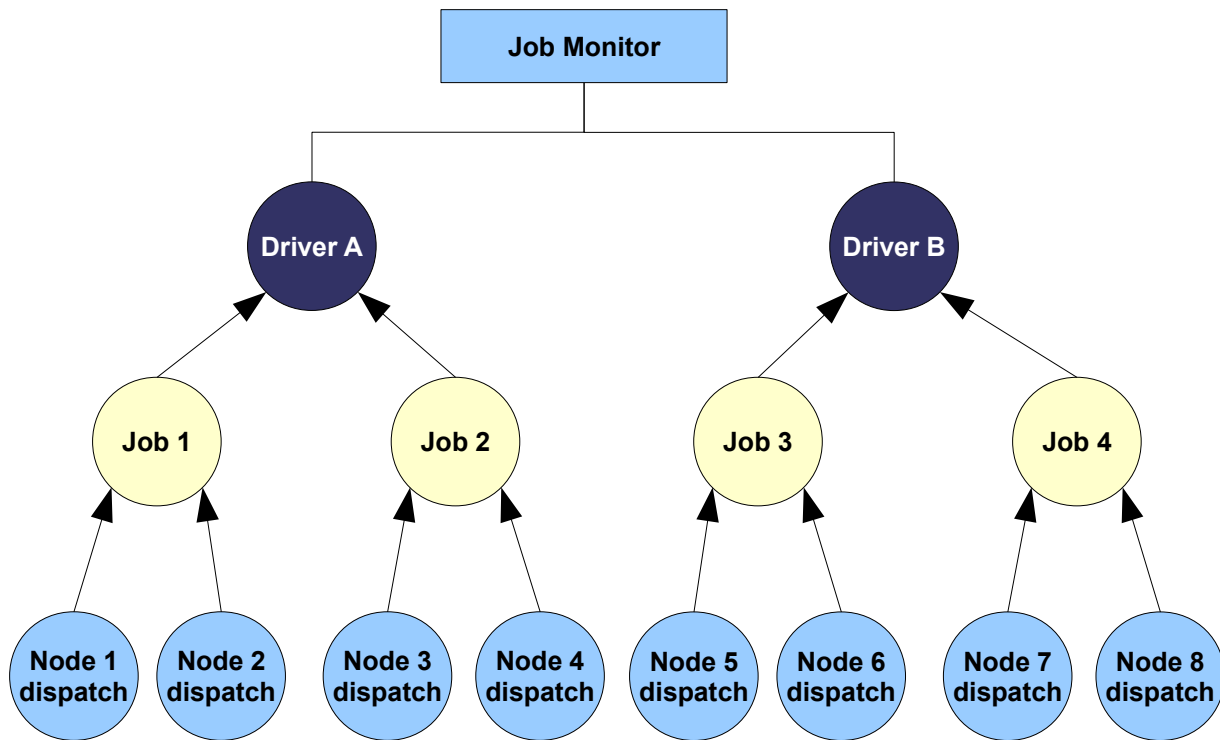
Active servers: 1 Active nodes: 2

4.15 Job monitoring API

In the same way that the topology monitoring API maintains a representation of the grid topology, the job monitoring API maintains a representation of the jobs being processed within the grid, including which jobs are processed by each drivers and which nodes the jobs are dispatched to.

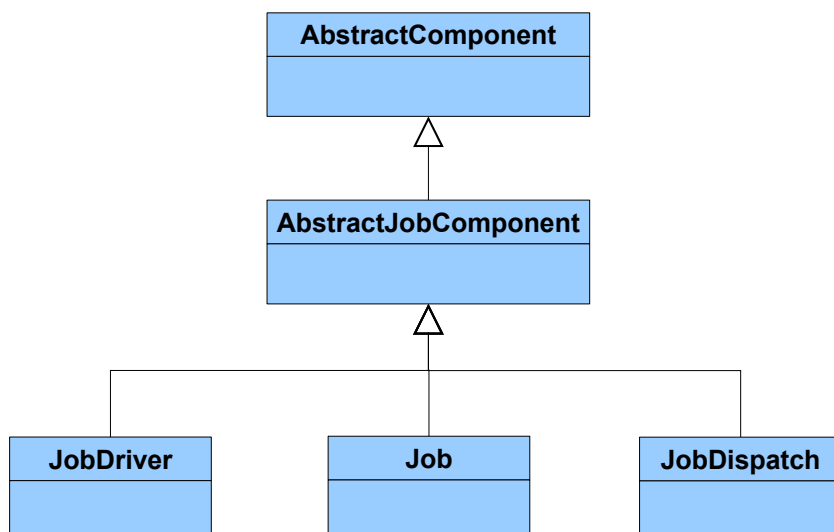
4.15.1 Job monitor and jobs hierarchy

The class [JobMonitor](#) maintains a representation of the jobs as a hierarchy with 3 levels: driver / job / job dispatch, as shown in this picture:



Note that a JPPF job can be submitted to multiple drivers, either in parallel from the same client, or from one driver to another in a multi-driver topology. [JobMonitor](#) provides an API to retrieve all the drivers a job is submitted to.

Each element of the jobs hierarchy has its own class, represented in this object model:



As we can see, job elements share a common super class with the topology elements: [AbstractComponent](#), which provides the base API to navigate a tree of elements identified by their uuid.

[JobMonitor](#) provides the following API to navigate the job hierarchy:

```
public class JobMonitor extends TopologyListenerAdapter {
    // Get the driver with the specified uuid
    public JobDriver getJobDriver(String driverUuid)

    // Get the drivers monitored by this job monitor
    public List<JobDriver> getJobDrivers()

    // Get the drivers to which a job was submitted
    public List<JobDriver> getDriversForJob(String jobUuid)

    // Get the dispatches of the specified job accrosss the entire topology
    // This returns job dispatches with nodes that may be connected to different servers
    public List<JobDispatch> getAllJobDispatches(String jobUuid)
}
```

The class [JobDriver](#) is essentially a wrapper for a [TopologyDriver](#) element, which represents a topology hierarchy that is orthogonal to the jobs hierarchy. It is defined as follows:

```
public class JobDriver extends AbstractJobComponent {
    // Get the proxy to the driver MBean that manages and monitors jobs
    public DriverJobManagementMBean getJobManager()

    // Get the associated driver from the topology manager
    public TopologyDriver getTopologyDriver()

    // Get a job handled by this driver from its uuid
    public Job getJob(final String jobUuid)

    // Get the list of jobs handled by this driver
    public List<Job> getJobs()
}
```

The [Job](#) class wraps a [JobInformation](#) object and provides information on the latest known state of a job, as seen by the driver it is submitted to. It is defined as follows:

```
public class Job extends AbstractJobComponent {
    // Get the information on the job
    public JobInformation getJobInformation()

    // Get the driver that holds this job
    public JobDriver getJobDriver()

    // Get the dispatch with the specified node uuid for this job
    public JobDispatch getJobDispatch(String nodeUuid)

    // Get the job dispatches for this job
    public List<JobDispatch> getJobDispatches()
}
```

[JobDispatch](#) represents a subset of a job that was sent to a node for execution and is the association of a [JobInformation](#) object with a [TopologyNode](#):

```
public class JobDispatch extends AbstractJobComponent {
    // Get the information on the node for ths job dispatch
    public TopologyNode getNode()

    // Get the information on the job
    public synchronized JobInformation getJobInformation()

    // Get the job to which this dispatch belongs
    public Job getJob()
}
```

4.15.2 Creating and configuring a job monitor

4.15.2.1 Constructors

To create and initialize a [JobMonitor](#), you have to call one of its two constructors:

```
public class JobMonitor extends TopologyListenerAdapter {
    // Create this job manager with the specified topology manager in
    // IMMEDIATE_NOTIFICATIONS mode
    public JobMonitor(TopologyManager topologyManager, JobMonitoringListener...listeners)

    // Create this job manager with the specified topology manager and event mode
    public JobMonitor(JobMonitorUpdateMode updateMode, long period,
        TopologyManager topologyManager, JobMonitoringListener...listeners)
}
```

Examples:

```
JPPFClient client = new JPPFClient();
TopologyManager tManager = new TopologyManager(client);
JobMonitoringListener listener1 = new JobMonitoringListenerAdapter() { ... };
JobMonitoringListener listener2 = new JobMonitoringListenerAdapter() { ... };
JobMonitoringListener[] listenerArray = { listener1, listener2 };

JobMonitor monitor;
monitor = new JobMonitor(tManager);
monitor = new JobMonitor(tManager, listener1);
monitor = new JobMonitor(tManager, listenerArray);
monitor = new JobMonitor(UpdateMode.IMMEDIATE_NOTIFICATIONS, -1L, tManager);
monitor = new JobMonitor(UpdateMode.POLLING, 1000L, tManager, listener1, listener2);
```

Remarks:

1) A call to:

```
new JobMonitor(topologyManager, listenerArray)
```

is equivalent to:

```
new JobMonitor(UpdateMode.IMMEDIATE_NOTIFICATIONS, -1L, topologyManager, listenerArray)
```

2) The vararg parameter `listeners` is optional and needs not be specified if you do not wish to register a job monitor listener at construction time.

3) As we can see, a [JobMonitor](#) requires a [TopologyManager](#) to work with, since it will use its [TopologyDriver](#) and [TopologyNode](#) objects to construct the [JobDriver](#) and [JobDispatch](#) instances in the jobs hierarchy.

4.15.2.2 Update modes

As we have seen, a [JobMonitor](#) is given an update mode at construction time, whether implicitly or explicitly. The update mode is one of the elements of the [JobMonitorUpdateMode](#) enum, defined as follows:

```
public enum JobMonitorUpdateMode {
    // Updates are computed by polling job information from the drivers
    // at regular intervals
    POLLING,

    // Updates are computed from JMX notifications and pushed immediately
    // as job monitor events. This means one event for each jmx notification
    IMMEDIATE_NOTIFICATIONS,

    // Updates are computed from JMX notifications and published periodically as job
    // monitor events. Notifications are merged/aggregated in the interval between
    // publications
    DEFERRED_NOTIFICATIONS
}
```

In POLLING and DEFERRED_NOTIFICATIONS modes, updates are published periodically as events, hence the need for the period parameter in the constructor, which represents the interval between publications in milliseconds. There is however a significant difference between these two modes: in POLLING mode, the period represents the interval between two JMX requests to the driver(s). Since this involves network communication, it can be a slow operation and the value for period should not be too small, otherwise the job monitor may not be able to cope. A period of 1000L (1 second) or more is recommended. In DEFERRED_NOTIFICATIONS mode, the information to publish is already present in memory and the period represents the interval between two publications as events. Thus the period can be significantly smaller than in POLLING mode. For instance, when displaying the updates in a desktop GUI, you might want to make sure that the user's eyes will perceive them as continuous updates. For this you need to have at least 25 updates per second, which means a period of 40 milliseconds or less.

In IMMEDIATE_NOTIFICATIONS mode, updates are immediately pushed as events, therefore no period parameter is needed and the value used in the constructor is simply ignored.

To conclude, the purpose of the update modes is to provide options for the tradeoff between the accuracy of the job updates and the ability for the application to cope with the generated workload. For instance, a GUI application may not need to display all the updates, in which case the POLLING mode would be sufficient. If you need to receive all updates, then IMMEDIATE_NOTIFICATIONS is the mode to use. Finally, please also note that POLLING is the mode which generates the least network traffic.

4.15.3 Receiving job monitoring events

[JobMonitor](#) allows registering one or more listeners to job events, either from its constructors, as we have seen in the previous section, or after construction time, with the following API:

```
public class JobMonitor extends TopologyListenerAdapter {
    // Initialize with the specified topology manager and listeners
    public JobMonitor(TopologyManager topologyManager, JobMonitoringListener...listeners)

    // Initialize with the specified topology manager, event mode and listeners
    public JobMonitor(UpdateMode updateMode, long period, TopologyManager topologyManager,
        JobMonitoringListener...listeners)

    // Add a listener to the events emitted by this job monitor
    public void addJobMonitoringListener(JobMonitoringListener listener)

    // Remove a listener to the events emitted by this job monitor
    public void removeJobMonitoringListener(JobMonitoringListener listener)
}
```

A listener is an implementation of the [JobMonitoringListener](#) interface, which provides the following notification methods:

```
public interface JobMonitoringListener extends EventListener {
    // Called when a new driver is added to the topology
    void driverAdded(JobMonitoringEvent event);

    // Called when a new driver is added to the topology
    void driverRemoved(JobMonitoringEvent event);

    // Called when a job is added to the driver queue
    void jobAdded(JobMonitoringEvent event);

    // Called when a job is removed from the driver queue
    void jobRemoved(JobMonitoringEvent event);

    // Called when the state a job has changed
    void jobUpdated(JobMonitoringEvent event);

    // Called when a job is dispatched to a node
    void jobDispatchAdded(JobMonitoringEvent event);

    // Called when a job dispatch returns from a node
    void jobDispatchRemoved(JobMonitoringEvent event);
}
```

If you do not need to implement all the methods, you may instead extend the [JobMonitoringListenerAdapter](#) class, which provides an empty implementation of each method in [JobMonitoringListener](#).

These methods provide an input parameter of type [JobMonitoringEvent](#):

```
public class JobMonitoringEvent extends EventObject {
    // Get the job monitor which emitted this event
    public JobMonitor getJobMonitor()

    // Get the job driver from which this event originates
    public JobDriver getJobDriver()

    // Get the related job, if any
    public Job getJob()

    // Get the related job dispatch, if any
    public JobDispatch getJobDispatch()
}
```

Note that `getJob()` and `getJobDispatch()` may return null, depending on the notification method called:

- `getJob()` will return null for the `driverAdded()` and `driverRemoved()` notifications
- `getJobDispatch()` only returns a value for the `jobDispatchAdded()` and `jobDispatchRemoved()` notifications.

As an example, the following code sample prints all the job monitoring events to the system console:

```
public class MyJobMonitoringListener implements JobMonitoringListener {
    @Override public void driverAdded(JobMonitoringEvent event) {
        print("driver ", event.getJobDriver(), " added");
    }

    @Override public void driverRemoved(JobMonitoringEvent event) {
        print("driver ", event.getJobDriver(), " removed");
    }

    @Override public void jobAdded(JobMonitoringEvent event) {
        print("job ", event.getJob(), " added to driver ", event.getJobDriver());
    }

    @Override public void jobRemoved(JobMonitoringEvent event) {
        print("job ", event.getJob(), " removed from driver ", event.getJobDriver());
    }

    @Override public void jobUpdated(JobMonitoringEvent event) {
        print("job ", event.getJob(), " updated");
    }

    @Override public void jobDispatchAdded(JobMonitoringEvent event) {
        print("job ", event.getJob(), " dispatched to node ", event.getJobDispatch());
    }

    @Override public void jobDispatchRemoved(JobMonitoringEvent event) {
        print("job ", event.getJob(), " returned from node ", event.getJobDispatch());
    }

    private void print(Object...args) {
        StringBuilder sb = new StringBuilder();
        for (Object o: args) {
            if (o instanceof AbstractComponent)
                sb.append(((AbstractComponent) o).getDisplayName());
            else sb.append(o);
        }
        System.out.println(sb.toString());
    }
}

TopologyManager tManager = ...;
// register the listener with the job monitor
JobMonitor monitor = new JobMonitor(tManager, new MyJobMonitoringListener());
// or, later on
monitor.addJobMonitoringListener(new MyJobMonitoringListener());
```

4.16 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 instance the number of nodes connected to a server, or the number of jobs in the server queue, etc.

[JPPFSnapshot](#) exposes the following API:

```
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();
    // Get the time elapsed between the creation of this snapshot and its last update
    double getLastUpdateNanos();
}
```

The label of a snapshot is expected to be unique and enables identifying 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:

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

Since it implements [Iterable<JPPFSnapshot>](#), a [JPPFStatistics](#) object can be used directly in a for loop or a `forEach` statement:

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

// forEach() with a lambda expression
stats.forEach(snapshot -> System.out.println("got '" + snapshot.getLabel() + "'"));
```

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

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

It can also be obtained remotely via the management APIs, as described in the section [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)  
}
```

The `getLocalizedLabel()` methods provide a short, localized description of what the snapshot is.

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

4.17 The Location API

4.17.1 Definition

This API allows developers to easily write data to, or read data from various sources: JVM heap, file system, URL or Maven central artifacts. It is based on the [Location](#) interface, which provides the following methods:

```
public interface Location<T> {  
    // Copy the content at this location to another location  
    <V> Location<V> copyTo(Location<V> 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;  
}
```

4.17.2 Built-in implementations

4.17.2.1 FileLocation

[FileLocation](#) represents a path in the file system.

Usage examples:

```
// create a location pointing to the source file:  
Location<String> src = new FileLocation("/home/user/docs/some_file.txt");  
// copy the source file to a new location and get the new location  
Location<String> dest = src.copyTo(new FileLocation("/home/user/other_file.txt"));
```

4.17.2.2 URLLocation

[URLLocation](#) can be used to get data to and from a URL, including HTTP and FTP URLs

Example:

```
// create a location pointing to a file on an FTP server  
Location<URL> src = new URLLocation("ftp://user:password@ftp.host.org/path/myFile.txt");  
// download the file from the FTP server to the local file system  
Location<String> dest = src.copyTo(new FileLocation("/home/user/destFile.txt"));
```

4.17.2.3 MavenLocation

[MavenLocation](#) is an extension of [URLLocation](#) which allows downloading Maven artifacts from a Maven repository. Contrary to other [Location](#) implementations, this one does not permit uploading data. To enforce this, its `getOutputStream()` method will always throw an [UnsupportedOperationException](#).

Examples:

```
// URL to a local repository on the file system  
String repositoryURL = "file:///home/me/.m2/repository";  
  
// a location that points to a Maven artifact with default "jar" packaging in the local repository  
Location<URL> jar = new MavenLocation(repositoryURL, "org.jppf:jppf-client:6.0-SNAPSHOT");  
// copy the artifact to the file system  
jar.copyTo(new FileLocation("lib/jppf-client-5.2.9.jar"));  
  
// create a location that points to a Maven artifact with "war" packaging  
Location<URL> war = new MavenLocation(repositoryURL, "org.jppf:jppf-admin-web:6.0", "war");  
// copy the artifact to the file system  
jar.copyTo(new FileLocation("tomcat-7.0/webapps/JPPFWebAdmin.war"));
```

Note: instances of [MavenLocation](#) only allow access to a single artifact and do not in any way handle Maven transitive dependencies.

4.17.2.4 MavenCentralLocation

[MavenCentralLocation](#) is a convenience [Location](#) implemented as a specialization of [MavenLocation](#), where the URL of the Maven repository is implicitly specified as that of the Maven Central repository. As for [MavenLocation](#), it does not permit uploading artifacts to the repository.

Examples:

```
// create a location that points to a Maven artifact with default "jar" packaging
Location<URL> jar = new MavenCentralLocation("org.jpff:jpff-client:5.2.9");
// copy the artifact to the file system
jar.copyTo(new FileLocation("lib/jpff-client-5.2.9.jar"));

// create a location that points to a Maven artifact with "war" packaging
Location<URL> war = new MavenCentralLocation("org.jpff:jpff-admin-web:6.0", "war");
// copy the artifact to the file system
jar.copyTo(new FileLocation("tomcat-7.0/webapps/JPPFWebAdmin.war"));
```

4.17.2.5 MemoryLocation

[MemoryLocation](#) represents a block of data in memory that can be copied from or sent to another location.

Example:

```
MyClass object = ...;
try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
     ObjectOutputStream oos = new ObjectOutputStream(baos)) {
    // serialize the object
    oos.writeObject(object);
    oos.flush();
    // store the serialized object in a memory location
    Location<byte[]> dataLocation = new MemoryLocation(baos.toByteArray());
    // copy the serialized object to a file
    dataLocation.copyTo(new FileLocation("/home/user/store/object.ser"));
} catch (Exception e) {
    e.printStackTrace();
}
```

4.17.3 Related reference

The location API is notably used to [specify a classpath in a job SLA](#).

4.18 Job selectors

Many of the methods in [JPPFClient](#) and in the job management interface [DriverJobManagementMBean](#) use a [JobSelector](#), which allows filtering the jobs in the client's or driver's queue according to user-defined criteria. This interface is defined as follows:

```
public interface JobSelector extends Serializable {
    // Whether the specified job is accepted by this selector
    boolean accepts(JPPFDistributedJob job);

    // Negate this job selector
    default JobSelector negate()

    // Obtain a job selector that realizes a logical "and" operation on this selector and an other
    default JobSelector and(final JobSelector other) throws NullPointerException

    // Obtain a job selector that realizes a logical "or" operation on this selector and an other
    default JobSelector or(final JobSelector other) throws NullPointerException

    // Obtain a job selector that realizes a logical "xor" operation on this selector and an other
    default JobSelector xor(final JobSelector other) throws NullPointerException
}
```

As we can see, it acts as a yes/no filter for instances of [JPPFDistributedJob](#), which represent client-side or server-side jobs. Job selectors are similar in functionality to the [node selectors](#) seen later in this documentation. Job selectors can also be composed into complex predicates, by the means of the logical operators "and", "or", "xor" and "negate".

The JPPF API provides a number of predefined implementations of [JobSelector](#), which perform comparisons or tests on the data provided by the [JPPFDistributedJob](#) interface: job uuid, name, [SLA](#) and [metadata](#). Many of these selectors can work with or without a metadata key. When a metadata key is not specified, the selector will apply to the jobs' names.

4.18.1 Logical job selectors

As mentioned previously, job selectors can be composed into more complex predicates, using the boolean operators "and", "or", "xor" and "negate". they can be used as follows:

```
// matches jobs whose value for metadata key "key1" is "value1"
JobSelector selector1 = new EqualJobSelector("key1", "value1");
// selects jobs whose name matches the regex .*my_job.*
JobSelector selector2 = new RegexJobSelector(".*my_job.*");

// compose the selectors using logical operators
JobSelector andSelector = selector1.and(selector2);
JobSelector orSelector = selector1.or(selector2);
JobSelector xorSelector = selector1.xor(selector2);
JobSelector notSelector2 = selector2.negate();
```

4.18.2 AllJobsSelector

An instance of [AllJobsSelector](#) will select all the jobs present in the driver at the time a management request is made. To use it, you can either construct a new instance or use the predefined constant [JobSelector.ALL_JOBS](#). For example:

```
DriverJobManagementMBean jobManager = ...;
// get information on all the jobs
JobInformation[] jobInfos = jobManager.jobInformation(new AllJobsSelector());
// alternatively, use the ALL_JOBS constant
jobInfos = jobManager.jobInformation(JobSelector.ALL_JOBS);
```

4.18.3 Job uuid and name selectors

Instances of [JobUuidSelector](#) and [JobNameSelector](#) filter jobs based on a set of job uuids or job names, provided in one of their constructors. Example usage:

```
// create a selector with known uuids
JobSelector uuidSelector = new JobUuidSelector("job_uuid_1", "job_uuid_2");
// create a job name selector from a list of names
List<String> names = new ArrayList<>();
for (JPPFjob job: jobs) names.add(job.getName());
JobSelector nameSelector = new JobNameSelector(names);
```

4.18.4 Binary comparison selectors

The selectors [AtLeastJobSelector](#), [AtMostJobSelector](#), [EqualsJobSelector](#), [LessThanJobSelector](#), [MoreThanJobSelector](#) and [NotEqualsJobSelector](#) respectively perform the equivalent of the mathematical comparisons \geq , \leq , $=$, $<$, $>$ and \neq . However, they apply to any [Comparable](#) value, so they are not limited to numbers.

These selectors can be applied to either a job metadata value, if a key is specified, or to a job name otherwise.

Here are some examples:

```
JobSelector selector;
// value of metadata "my.counter" should be >= 5
selector = new AtLeastJobSelector("my.counter", 5);
// "my.date" metadata should be no later than 5 minutes from now
selector = new AtMostJobSelector("my.date", ZonedDateTime.now().plusMinutes(5));
// job name should be equal to "first job"
selector = new EqualsJobSelector("first job");
// enums are comparable too
selector = new LessThanJobSelector("jobType", MyJobType.PROD);
// the "ratio" metadata should be more than the double value 0.65
selector = new MoreThanJobSelector("ratio", 0.65d);
// "my.file" must not point to "/opt/app/config.yml"
selector = new NotEqualsJobSelector("my.file", new File("/opt/app/config.yml"));
```

4.18.5 Range comparison operators

The selectors [BetweenEEJobSelector](#), [BetweenEIJobSelector](#), [BetweenIEJobSelector](#) and [BetweenIJJobSelector](#) respectively perform the equivalent of the range comparisons $a < x < b$, $a < x \leq b$, $a \leq x < b$ and $a \leq x \leq b$. Similarly to binary comparisons, they work with any [Comparable](#) value and may apply to a job name, or a metadata value if its key is specified.

Examples:

```
JobSelector selector;
// "amount" must be more than 100.0 and less than 500.0
selector = new BetweenEEJobSelector("amount", 100d, 500d);
// "date" must be after 2 hours ago and no later than 45 minutes from now
ZonedDateTime now = ZonedDateTime.now();
selector = new BetweenEIJobSelector("date", now.minusHours(2), now.plusMinutes(45));
// job name must be at least "job1" and less than "job2". Example: "job12-hello"
selector = new BetweenIEJobSelector("job1", "job2");
// "day.of.week" must be from monday to wednesday included
selector = new BetweenIJJobSelector("day.of.week", DayOfWeek.MONDAY, DayOfWeek.WEDNESDAY);
```

4.18.6 Contains, one of and regex job selectors

These selectors perform the following boolean functions:

- [ContainsJobSelector](#): determines whether a job name or metadata value contains a substring. If the value of the specified metadata is not a string, the result is false.
- [IsOneOfJobSelector](#): determines whether is one of a collection or array of values, based on `equals()` comparisons.
- [RegexJobSelector](#): applies a regex to a job name or metadata value and returns whether it matches or not. Only applies to string values. If the value of the specified metadata is not a string, false is returned.

Some usage examples:

```
JobSelector selector;
// the metadata "label" must have a substring "test string"
selector = new ContainsJobSelector("label", "test string");
// the metadata "dayOfMonth" must be either 1, 7, 14 or 28
selector = new IsOneOfJobSelector("dayOfMonth", 1, 7, 14, 28);
// the job name must have a substring "test" (case insensitive) followed by at least one digit
selector = new RegexJobSelector(".*test[0-9]+.*", Pattern.CASE_INSENSITIVE);
```

4.18.7 Scripted job selector

A [ScriptedJobSelector](#) uses a script expression that evaluates to a boolean to determine which jobs are accepted. The script language must be accepted by the JVM and must conform to the [JSR-223](#) / [javax.script](#) API specifications.

A scripted job selector is particularly useful when you need to implement a complex filtering logic but do not want to deploy the associated code in the driver's classpath.

[ScriptedJobSelector](#) extends [BaseScriptEvaluator](#) and is defined as follows:

```
public class ScriptedJobSelector extends BaseScriptEvaluator implements JobSelector {
    // Initialize this selector with the specified language and script
    public ScriptedJobSelector(String language, String script)

    // Initialize this selector with the specified language and script reader
    public ScriptedJobSelector(String language, Reader scriptReader) throws IOException

    // Initialize this selector with the specified language and script file
    public ScriptedJobSelector(String language, File scriptFile) throws IOException

    @Override
    public boolean accepts(final JPPFDistributedJob job)
}
```

The script will be evaluated for each [JPPFDistributedJob](#) passed as input to the `accepts()` method of the selector. The job can be accessed from the script using a predefined variable named "jppfJob", as in this example:

```
DriverJobManagementMBean jobManager = ...;
// create a selector with a Javascript script
String script = "jppfJob.getName().startsWith('MyJob-')";
JobSelector selector = new ScriptedJobSelector("javascript", script);
// or an equivalent Groovy script
selector = new ScriptedJobSelector("groovy", "return " + script);
// get information on the selected jobs
JobInformation[] jobInfos = jobManager.jobInformation(selector);
```

4.18.8 Custom job selector

When a job selector requires a complex logic and its performance is critical, you can always write your own implementation of [JobSelector](#), as in this example:

```
public class CustomJobSelector implements JobSelector {
    @Override
    public boolean accepts(JPPFDistributedJob job) {
        // retrieve a parameter from the job metadata
        int param = job.getMetadata().getParameter("my.param", -1);
        // use the param in the filtering expression
        return !job.getSLA().isSuspended() && (param >= 0);
    }
}
```

When the job selector is evaluated by the driver, the code of its implementation, along with all the classes it depends on, must be deployed in the driver's classpath. This is usually done by dropping the class folder or jar file that contains the code in the driver's `/lib` folder.

4.19 Job dependencies and job graphs

4.19.1 Introduction

JPPF jobs can depend on other jobs, meaning that their execution will start only when all the jobs they depend on have completed. In turn, the jobs they depend on may have their own dependencies, and the same behavior is applied recursively.

4.19.1.1 Definitions

Together, a set of jobs and their dependencies form a **job dependency graph**, which is a [directed acyclic graph](#) (DAG). In the rest of this documentation such graphs may be shortened to **job graphs**. In JPPF terminology, each job in a dependency graph is called a **node** or **graph node**.

A job dependency graph distinguishes between two kinds of nodes:

- **graph roots** represent jobs no other job depends on. Upon completion of a job graph root, it and all its dependencies are removed from the graph
- **regular nodes** represent jobs such that at least one other job depends on them

Nodes in a job dependency graph are identified with a **dependency id**. This id is an arbitrary identifier string, distinct from the job's UUID, which fulfills two major roles:

- when specified, it marks the job as being part of a job dependency graph. Otherwise, it is just a regular job
- it uniquely identifies a job within a job graph and each dependency of a job is specified with its own dependency id.

According to this, a job dependency graph can thus be represented as a graph of dependency ids.

4.19.1.2 Features and limitations

The choice of an arbitrary dependency id implies that you can define job graphs even when not all the jobs in the graph already exist. All that's needed are the dependency ids of the jobs. In particular, you are not constrained to using job UUIDs, which are automatically created by JPPF, and thus cannot be known in advance.

Thanks to this level of freedom, a job graph can be submitted in multiple parts from multiple JPPF clients, including clients running on separate physical or virtual machines.

Furthermore, there is no constraint on the order of submission of the jobs in a dependency graph. The specification of each job always contains enough information to know what to do with it: either wait until all its dependencies have completed, or execute the job immediately.

The default behavior, when a job in a dependency graph is cancelled, is to cascade the cancellation to all the jobs that depend on it, recursively, until all the paths that lead to this job from the graph roots are cancelled. This behavior can be overridden by setting the `cascadeCancellation` attribute of each job's [dependency specification](#).

There is one major limitation, which applies to all multi-server grid topologies: all the jobs in a dependency graph must be submitted to the same JPPF driver. If the jobs of a graph are distributed over multiple drivers, then the graph may never complete in any of the drivers. In multi-server topologies, individual jobs in a graph may be offloaded to other drivers by the job scheduler, but only as regular jobs, that is, not a part of a graph. In other words, a job graph has a single, non-redundant representation held by a single driver at any given time. One way to address this is to assign an [execution policy](#) to the [client-side SLA](#) of each job, which enforces submission to the same driver.

4.19.2 Specifying job dependencies

4.19.2.1 With the SLA's dependency specification

We have seen, in section [job dependencies specification](#), that a job dependency graph can be specified using the `jobDependencySpec` attribute of the server-side SLA in each job. Let's take the example of a simple graph with two jobs, where `job_1` depends on `job_2`:

```
// define the dependency graph
job_1.getSLA().getDependencySpec()
    .setId(job_1.getName())
    .setGraphRoot(true)                // add job_1 to the graph as a root
    .addDependencies(job_2.getName()); // add job_2 as a dependency of job_1
job_2.getSLA().getDependencySpec()
    .setId(job_2.getName());
```

Here, we have defined a job graph where the dependency ids are equal to the job names, job_1 is a graph root and job_2 is a dependency of job_1. It is also implied that the cancellation of job_2 will cascade to the cancellation of job_1, since it is the default behavior.

4.19.2.2 With the JPPFJob API

Defining a job graph with the [JobDependencySpec](#) API can be quite cumbersome, especially when defining complex job graphs. To alleviate this burden, the [JPPFJob](#) class provides a number of methods that make the graph definition easier, shorter and more readable:

```
public class JPPFJob extends AbstractJPPFJob<JPPFJob>
    implements Iterable<Task<?>>, Future<List<Task<?>>> {

    // Set this job's dependency id from the specified id supplier
    public JPPFJob setDependencyId(Supplier<String> idSupplier)

    // Set this job's dependency id from a dependency id supplier
    public JPPFJob setDependencyId(JobDependencyIdSupplier idSupplier)

    // Set this job's dependency id
    public JPPFJob setDependencyId(String id)

    // Convenience method to set this job's uuid as dependency id
    public JPPFJob setUuidAsDependencyId()

    // Convenience method to set this job's name as dpendency id
    public JPPFJob setNameAsDependencyId()

    // Add the specified array of jobs as dependencies to this job
    public JPPFJob addDependencies(JPPFJob...dependencies)

    // Add the specified collection of jobs as dependencies to this job
    public JPPFJob addDependencies(Collection<JPPFJob> dependencies)

    // Set whether this job is a root in a job dependency graph
    public JPPFJob setGraphRoot(boolean graphRoot)

    // Set whether cancellation of this job should trigger the cancellation of its dependents
    public JPPFJob setCascadeCancellation(boolean cascadeCancellation)
}
```

Notice the `setDependencyId()` methods that take either a generic [Supplier<String>](#) or [JobDependencyIdSupplier](#), which provide a lot of flexibility as to how the dependency ids are computed and supplied to the jobs. In the following code example, each line achieves the same goal, to set the job's name as its dependency id:

```
myJob.getSLA().getDependencySpec().setId(myJob.getName());
myJob.setDependencyId(() -> myJob.getName());
myJob.setDependencyId(job -> job.getName());
myJob.setDependencyId(myJob.getName());
myJob.setNameAsDependencyId();
```

With this API, the example in the [previous section](#) can be rewritten much more concisely:

```
// define the dependency graph
job_1.setNameAsDependencyId()
    .setGraphRoot(true) // add job_1 to the graph as a root
    .addDependencies(job_2); // add job_2 as a dependency of job_1
job_2.setNameAsDependencyId();
```

4.19.3 Cycles in the job dependency graph

In a job graph, cycles, or circular dependencies, can only be detected in the server where the graph is submitted. There are two reasons for this:

- since jobs in a graph can be submitted by multiple JPPF clients, only the server is expected to hold the full graph
- because of that, job dependencies cannot be fully identified until they are sent to the server. Only then can they be associated with an existing job UUID.

When a cycle is detected, the server will first throw and log a `JPPFJobDependencyCycleException` whose message describes the cycle path, for example:

```
org.jppf.node.protocol.graph.JPPFJobDependencyCycleException: job-3 ==> job-1 ==> job-2 ==> job-3
```

After this, the JPPF server will cancel all the jobs in the cycle path, following the specified [cancellation behavior](#).

4.19.4 Job cancellation behavior

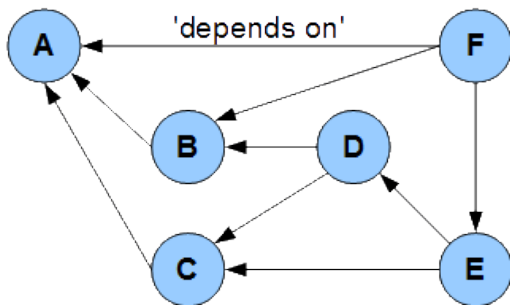
By default, when a job in a graph is cancelled, all the jobs that depend on it will also be cancelled. This process is repeated recursively until we reach all the graph roots whose dependency path leads to the originally cancelled job. This behavior is called cancellation cascading.

This can be overridden for each individual job, by setting the `cascadeCancellation` attribute of thier dependency specification to `false`:

```
JPPFJob job = ...;  
job.getSLA().getJobDependencySpec().setCascadeCancellation(false);
```

When cancellation cascading is turned of on a job, the jobs that depend on it will simply run to completion, instead being cancelled and temprinating early.

To illustrate this, consider the following job graph:



By default, if we cancel Job C, this will trigger the cancellation of jobs D, E and F, but not of jobs A and B which will run to completion. However, if cancellation cascade is turned off for job E, then only jobs D and E will be cancelled as a consequence of cancelling job C.

4.19.5 Job graphs monitoring

Job graphs in the JPPF server can be monitored with a specialized management MBean which implements the [JobDependencyManagerMBean](#) interface:

```
public interface JobDependencyManagerMBean {  
    // Object name of this server-side MBean  
    String MBEAN_NAME = "org.jppf:name=jobDependencyManager,type=driver";  
  
    // Get the size (number of nodes or vertices that represent jobs) of the job dependency graph  
    int getGraphSize();  
  
    // Get the ids of all the nodes currently in the graph  
    Set<String> getNodeIds();  
  
    // Get all the nodes in the job dependency graph  
    Collection<JobDependencyNode> getAllNodes();  
  
    // Get the nodes in the job dependency graph whose corresponding job is queued in the server  
    Collection<JobDependencyNode> getQueuedNodes();  
  
    // Get the nodes in the job dependency graph whose corresponding job is queued in the server,  
    // filtered by the specified job selector  
    Collection<JobDependencyNode> getQueuedNodes(JobSelector selector);  
  
    // Get the node with the specified dependency id  
    JobDependencyNode getNode(String id);  
  
    // Get the graph of job dependencies  
    JobDependencyGraph getGraph();  
}
```

With this MBean, you can retrieve the full job dependency graph with the `getGraph()` method. This method returns an instance of the [JobDependencyGraph](#) interface, defined as follows:


```

public interface JobDependencyGraph extends Serializable {
    // Get the node for the specified dependency id
    JobDependencyNode getNode(String id);
    // Get the size of this graph, that is, the number of nodes or vertices
    int getSize();
    // Get the dependency ids of all the nodes currently in this graph
    Set<String> getNodeIds();
    // Get the node whose corresponding job has the specified uuid
    JobDependencyNode getNodeByJobUuid(String jobUuid);
    // Get all the nodes currently in this graph
    Collection<JobDependencyNode> getAllNodes();
    // Get the nodes in the job dependency graph whose corresponding job is queued in the server
    Collection<JobDependencyNode> getQueuedNodes();
    // Get the nodes that depend on the node with the specified dependency id
    Collection<JobDependencyNode> getDependedOn(String id);
    // Determine whether the job dependency graph is empty
    boolean isEmpty();
}

```

Each node in the graph is represented as an instance of [JobDependencyNode](#):

```

public class JobDependencyNode implements Serializable {
    // Get the dependency id of this node
    public String getId()
    // Determine whether this node has any pending (not completed) dependency
    public boolean hasPendingDependency()
    // Get the dependency with the specified id
    public JobDependencyNode getDependency(String id)
    // Get the dependencies for this node
    public Collection<JobDependencyNode> getDependencies()
    // Get the ids of the dependencies for this node
    public Set<String> getDependenciesIds()
    // Get the nodes that depend on this node
    public Collection<JobDependencyNode> getDependedOn()
    // Determine whether the job represented by this node has completed
    public boolean isCompleted()
    // Get the JPPF uuid of the associated job
    public String getJobUuid()
    // Whether this node is a graph root
    public boolean isGraphRoot()
    // Determine whether the job represented by this dependency node has been cancelled
    // or should be cancelled when it arrives in the server queue
    public boolean isCancelled()
}

```

Here is an example usage of the MBean and related classes:

```

// get the job dependency manager MBean from a JPPF client
JPPFClient client = new JPPFClient();
JMXDriverConnectionWrapper jmx = client.awaitWorkingConnectionPool().awaitWorkingJMXConnection();
JobDependencyManagerMBean manager = jmx.getJobDependencyManager();

// get the job dependency graph and use it
JobDependencyGraph graph = manager.getGraph();
if (!graph.isEmpty()) {
    JobDependencyNode node = graph.getNode("my_dependency_id");
    if ((node != null) && !node.isCompleted()) {
        for (JobDependencyNode dependency: node.getDependencies()) {
            // do something with the job's dependencies ...
        }
    }
}
}

```

See also:

- [job SLA dependencies specification](#)
- [JMXDriverConnectionWrapper.getJobDependencyManager\(\)](#)

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 [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 properties is as specified in the [Java Properties class](#). To enable a JPPF component to retrieve these 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 file system is always looked up first.

5.2 Includes, substitutions and scripted values in the configuration

5.2.1 Includes

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 interpreted 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.

5.2.2 Substitutions in the values of configuration properties

The JPPF configuration can handle a syntax of the form “*propertyName* = *prefix****\${otherPropertyName}****suffix*”, where *prefix* and *suffix* are arbitrary strings and ***\${otherPropertyName}*** is a placeholder referencing another property, whose value will be substituted to the placeholder. If you have experience with Apache Ant, this syntax is very similar to the way Ant properties are used in a build script.

Let's take an example illustrating how this works. The following property definitions:

```
prop.1 = value1
prop.2 = ${prop.1}
prop.3 = value3 ${prop.2}
prop.4 = value4 ${prop.2} + ${prop.3}
```

will be resolved into:

```
prop.1 = value1
prop.2 = value1
prop.3 = value3 value1
prop.4 = value4 value1 + value3 value1
```

Note 1: the order in which the properties are defined has no impact on the resolution of substitutions.

Note 2: substitutions are resolved after all includes are fully resolved and loaded.

5.2.2.1 Unresolved substitutions

A referenced property is unresolvable if, and only if, it refers to a property that is not defined or it is involved in a resolution cycle. In this case, the value of the unresolved value will be the literal syntax in its initial definition, that is in the literal form “\${otherPropertyName}”. Let’s illustrate this with examples.

In the following configuration:

```
prop.1 = ${prop.2}
```

the value of “prop.1” will remain “\${prop.2}”, since the property “prop.2” is not defined.

In this configuration:

```
prop.1 = ${prop.2}
prop.2 = ${prop.3} ${prop.1}
prop.3 = value3
```

“prop.1” and “prop.2” introduce an unresolvable cycle, and only the reference to “prop.3” can be fully resolved. According to this, the final values will be:

```
prop.1 = ${prop.2}
prop.2 = value3 ${prop.1}
prop.3 = value3
```

5.2.2.2 Environment variable substitutions

Any reference to a property name in the form “env.variableName” will be substituted with the value of the environment variable whose name is “variableName”. For example, if the environment variable JAVA_HOME is defined as “/opt/java/jdk1.7.0”, then the following configuration:

```
prop.1 = ${env.JAVA_HOME}/bin/java
```

will be resolved into:

```
prop.1 = /opt/java/jdk1.7.0/bin/java
```

If the value of a property refers to an undefined environment variable, then the reference will be replaced with the literal syntax in its initial definition, that is in the literal form “\${env.<UndefinedVariable>}”.

5.2.2.3 System properties substitutions

References to system properties work in exactly the same way as for environment variables, except that the prefix for a system property reference is “sys.” instead of “env.”. The syntax for the reference is then “\${sys.systemPropertyName}”.

For example, if the Java command includes an option “-DTest.property=some.test.value”, then the following configuration property:

```
prop.1 = ${sys.Test.property}
```

will be resolved into:

```
prop.1 = some.test.value
```

5.2.3 Scripted property values

The values of configuration properties can be partially or completely computed as expressions written in any JSR 223-compliant dynamic script language. Such properties contain one or more expressions of the form:

```
my.property = $script:language:source_type{script_source}$
```

Where:

- *language* is the script language to use, such as provided by the javax.script APIs. It defaults to "javascript"
- *source_type* determines how to find the script, with possible values "inline", "file" or "url". It defaults to "inline"
- *script_source* is either the script expression, or its location, depending on the value of *source_type*:
 - if *source_type* = inline, then *script_source* is the script itself.
For example: `my.prop = $script:javascript:inline{"hello " + "world"}$`
 - if *source_type* = file, then *script_source* is a script file, looked up first in the file system, then in the classpath.
For example: `my.prop = $script:javascript:file{/home/me/myscript.js}$`
 - if *source_type* = url, then *script_source* is a script loaded from a URL.
For example: `my.prop = $script:javascript:url{file:///home/me/myscript.js}$`

If the language or source type are left unspecified, they will be assigned their default value. For instance the following patterns will all resolve in `language = 'javascript'` and `source_type = 'inline'` :

```
$script{ 2 + 3 }$
$script:{ 2 + 3 }$
$script::{ 2 + 3 }$
$script::inline{ 2 + 3 }$
$script:javascript{ 2 + 3 }$
$script:javascript:{ 2 + 3 }$
```

Note: the syntax for scripted values may be quite cumbersome. It is possible to alleviate it by using 'S' or 's' instead of "script", and the first character (case insensitive) of the source type instead of its full name. For instance, the following properties are equivalent and will resolve to the exact same value:

```
prop.1 = $script:js:file{/home/me/script.js}$
prop.2= $S:js:f{/home/me/script.js}$
```

The default script language can be specified with the property '`jppf.script.default.language`'. This property is always evaluated first, in case it is also expressed with script expressions (which can only be in javascript). If it is not specified explicitly, it will default to 'javascript'. For example, in the following:

```
# using javascript expression to set the default language to groovy
jppf.script.default.language = $script{ 'groo' + 'vy' }$
# inline expression using default language 'groovy'
my.property = $script{ return 2 + 3 }$
```

The value of '`my.property`' will evaluate to 5, resulting from the inline groovy expression '`return 2 + 3`'.

The scripts are evaluated after all includes and variable substitutions have been resolved. This will allow the scripts to use a variable binding for the Properties (or [TypedProperties](#) object) being loaded, with the best possible accuracy. For example, in the following:

```
prop.1 = hello world
prop.2 = $script:javascript{ thisProperties.getString('prop.1') + ' of scripting' }$
```

the value of '`prop.2`' will evaluate to 'hello world of scripting', which is the value of '`prop.1`' to which another string is concatenated. The predefined variable *thisProperty* is bound to the properties object being evaluated. Note that the same result can be achieved with a property substitution:

```
prop.1 = hello world
prop.2 = $script:javascript{ '${prop.1}' + ' of scripting' }$
```

Here, the substitution of `${prop.1}` is performed *before* the script evaluation, and must be enclosed within quotes to be parsed as a string literal inside the script.

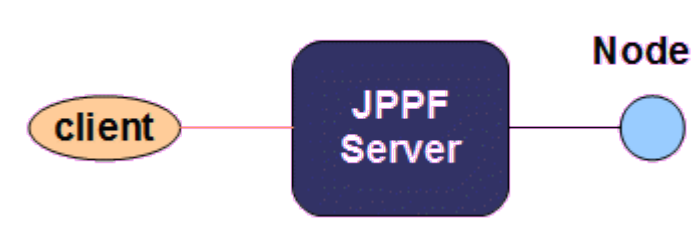
Finally, property values can contain any number of script expressions, which can be written in different script languages and loaded from different sources. For example, the value of the following property:

```
prop.1 = hello $script:javascript{'my ' + 'world'}$ number $script:groovy{return 2 + 3}$
```

will evaluate to 'hello my world number 5'.

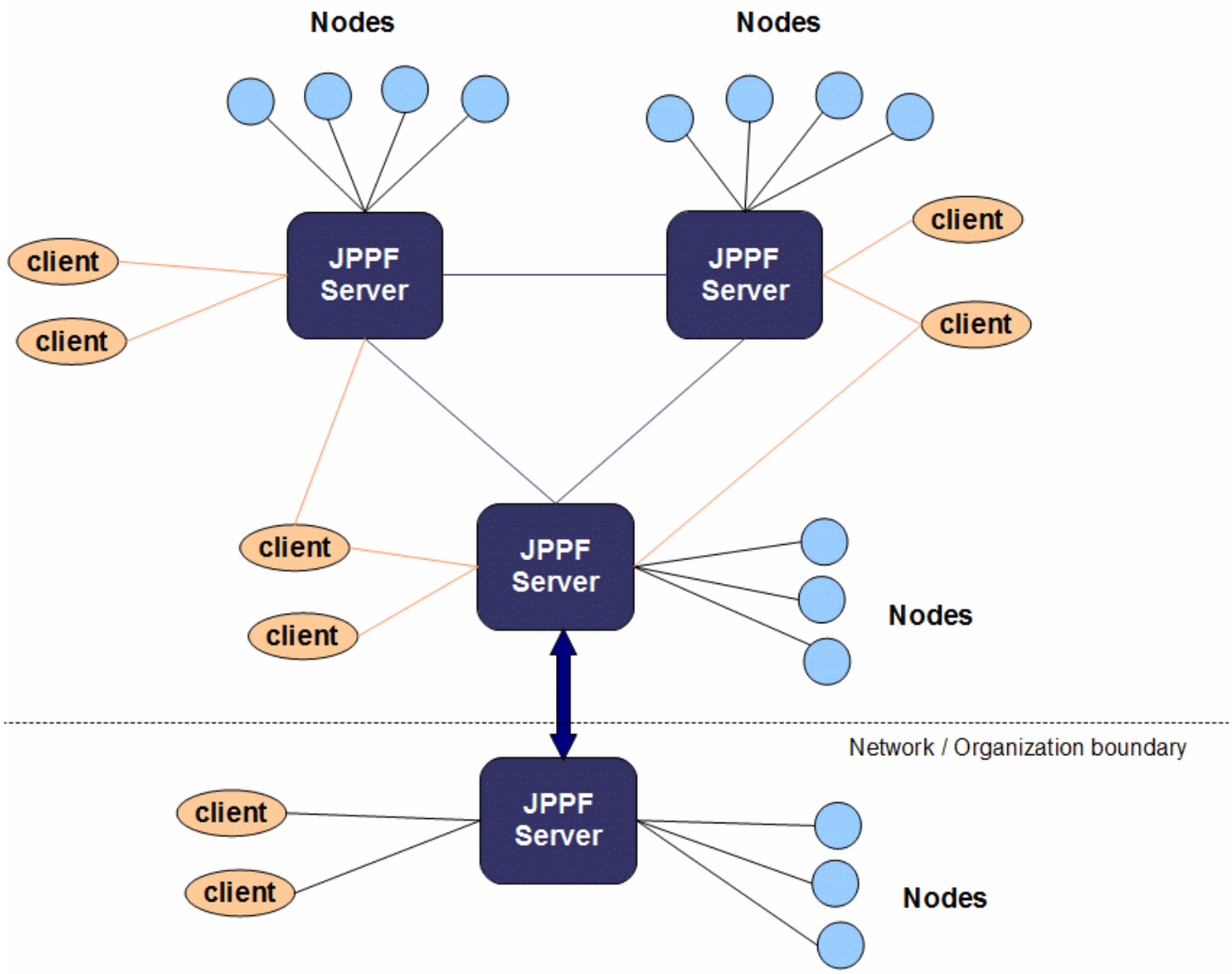
5.3 Reminder: JPPF topology

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, over plain connections, secure connections, or both. Each type of connection requires the configuration of a dedicated TCP port. In the configuration file, this property would be defined as follows, with its default value:

```
# JPPF server port for plain connections; default value is 11111
jppf.server.port = 11111

# JPPF server port for secure connections via SSL/TLS; default value is -1
jppf.ssl.server.port = 11143
```

Note 1: to disable either plain or secure connectivity, set the corresponding port value to -1.

Note 2: secure connectivity is disabled by default, therefore you must explicitly configure the secure port to enable it

Note 3: 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, since the port number will not be known in advance to connecting nodes and client

5.4.2 Server JVM options

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)
- location of Log4j configuration (-Dlog4j.configuration)
- 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 = -server -Xmx512m \
-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
```

This syntax allows configuring multiple paths in an OS-independent way, in particular with regards to the path separator character (e.g. ':' on Linux, ';' on Windows).

If a classpath element contains one or more spaces, the path(s) it defines must be surrounded with double quotes:

```
jppf.jvm.options = -Xmx512m -cp "dir with spaces/myJar1.jar" -cp NoSpaces/myJar2.jar
```


5.4.3 Specifying the path to the JVM

It is possible to choose which JVM will run a driver, by specifying the full path to the Java executable with the following property:

```
# Full path to the java executable
jppf.java.path = <path_to_java_executable>
# Linux example
jppf.java.path = /opt/jdk1.8.0/bin/java
# Windows example
jppf.java.path = C:/java/jdk1.7.0/bin/java.exe
```

This property is used by the shell script from the driver distribution that launches the driver (startDriver.sh or startDriver.bat) and when a driver is restarted with the [JPPFDriverAdminMBean.restartShutdown\(\)](#) management method or from the administration console.

5.4.4 Server discovery through UDP multicast

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.

5.4.4.1 Enabling and disabling UDP multicast

This is done with the following property, which defaults to true (enabled):

```
# Enable or disable broadcast of the JPPF driver's information s via UDP multicast
jppf.discovery.enabled = true
```

5.4.4.2 Configuration of UDP multicast

The configuration is done by defining a multicast group and port number, as in this example showing their default values:

```
# UDP multicast group to which the driver broadcasts its connection parameters
jppf.discovery.group = 230.0.0.1
# UDP multicast port to which the driver broadcasts its connection parameters
jppf.discovery.port = 11111
```

5.4.4.3 Broadcast interval

Since the UDP protocol offers no guarantee of delivery, the JPPF driver will periodically broadcast its connection information, at regular intervals defined with the following property:

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

5.4.4.4 Inclusion and exclusion patterns

The driver can be configured to allow or exclude broadcasting on specific network interfaces, according to their IP addresses. The following properties define inclusion and exclusion patterns for IPv4 and IPv6 addresses. Each of these patterns defines a list of comma- or semicolon- separated patterns. The IPv4 patterns can be expressed in either [CIDR](#) notation, or in a syntax defined in the Javadoc for the class [IPv4AddressPattern](#). Similarly, IPv6 patterns can be expressed in [CIDR](#) notation or in a syntax defined in [IPv6AddressPattern](#). This enables filtering out unwanted IP addresses: the discovery mechanism will only allow addresses that are included and not excluded.

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

Let's take for instance the following pattern specifications:

```
jppf.discovery.include.ipv4 = 192.168.1.
jppf.discovery.exclude.ipv4 = 192.168.1.128-
```

The equivalent patterns in CIDR notation would be:

```
jppf.discovery.include.ipv4 = 192.168.1.0/24
jppf.discovery.exclude.ipv4 = 192.168.1.128/25
```

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 192.168.1.128 ... 192.168.1.255. Thus, we actually defined a filter that only accepts addresses in the range 192.168.1.0 ... 192.168.1.127.

These 2 patterns can in fact be rewritten as a single inclusion pattern:

```
jppf.discovery.include.ipv4 = 192.168.1.-127
```

or, in CIDR notation:

```
jppf.discovery.include.ipv4 = 192.168.1.0/25
```

5.4.5 Connecting to other servers

We have seen in the "[reminder](#)" section 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). The benefit is that, when server A is connected to server B, B will be able to offload some of its workload to server A, for example when all nodes attached to B are already busy.

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

Because of this flexibility, it is possible to define any type of topology made of JPPF drivers, up to fully connected P2P topologies.

5.4.5.1 Orphan servers

The default behavior when a server doesn't have any attached node (i.e. orphan server) is, for the peer servers it is connected to, to not send any job to this server. Servers are automatically notified of the number of nodes attached to their peers, and will start sending jobs their way as soon as they have at least one node.

To force a server to send jobs to their peers even when they don't have any node, for instance if you wish to set a server as a router to other servers, you can set the following property in its configuration:

```
# ignore the fact that peer servers may not have any node; default to false
jppf.peer.allow.orphans = true
```

5.4.5.2 Configuring peer connections manually

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
jppf.peer.server_1.pool.size = 2
# enable heartbeat-based connection failure detection
jppf.peer.server_1.recovery.enabled = true

# connection to server_2
jppf.peer.server_2.server.host = host_2
jppf.peer.server_2.server.port = 11111
jppf.peer.server_2.pool.size = 2
```

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_2.server.port" must be equal to that of "jppf.server.port" in server_2's configuration.

As for auto-discovered servers, it is possible to specify the number of connections to each manually configured peer server with the "jppf.peer.<peer_name>.pool.size" property, which defaults to 1 if unspecified.

5.4.5.3 Discovering peer drivers via UDP multicast

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
# number of connections to establish with each discovered server, aka pool size
# (defaults to 1)
jppf.peer.pool.size = 2
# enable heartbeat-based connection failure detection
jppf.peer.recovery.enabled = true
```

For this to work, the server broadcast must be enabled on the peer server(s), and the properties defined in the previous "[server discovery](#)" section 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 "jppf.peer.discovery.enabled" is "false". Setting the default to "true" would cause each server to connect to all other servers accessible on the network, with a high risk of unwanted side effects.

It is also possible to define more than one connection with each discovered peer driver, by setting the property "jppf.peer.pool.size" to the desired number of connections. If this property is unspecified, it will default to 1.

5.4.5.4 Using manual configuration and server discovery together

It is possible to use the manual configuration together with the UDP multicast 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
# specify both discovery and manually configured drivers
jppf.peers = jppf_discovery server_1
# connection to server_1
jppf.peer.server_1.server.host = host_1
jppf.peer.server_1.server.port = 11111
```

5.4.5.5 Peer drivers load-balancing threshold

It is possible to configure a driver, such that it will start load-balancing its workload to other drivers only when it has less than a specified number of attached nodes. This is done with the following configuration property:

```
# Load-balance to peer drivers when the number of connected nodes is less than 3
jppf.peers.load.balance.threshold = 3
```

The default value of this property is [Integer.MAX_VALUE](#), which is the closest equivalent to an infinite threshold. This default value means that the driver will *always* load-balance to other drivers. On the other hand, a value of 1 or less means that the driver will *never* load-balance to other peer drivers, which is another way of saying that the other drivers are there for failover only.

5.4.6 JMX management configuration

JPPF uses JMX to provide remote management capabilities for the servers, and uses its own [JMX connector](#) for communication. The management features are enabled by default; this behavior can be changed by setting this property:

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

5.4.7 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, or job, sent by the client application. Each set of tasks sent to a node is called a "task 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.

5.4.7.1 General configuration

The algorithm to use is configured with the following property:

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

The algorithm name can be one of those predefined in JPPF, or a user-defined one. JPPF has a number of predefined load-balancing algorithms to compute the distribution of tasks to the nodes, each with its own configuration parameters.

The predefined possible values for the property `jppf.load.balancing.algorithm` are: `manual`, `autotuned`, `proportional`, `rl2` and `nodethreads`. If not specified, the algorithm defaults to `manual`. For example:

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

Each algorithm uses its own set of parameters, which define together a *profile* for the algorithm, A profile 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 simply 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.

5.4.7.2 Predefined algorithms

5.4.7.2.1 "manual" algorithm

With this algorithm, each bundle has a fixed number of tasks, meaning that each node will receive at most this number of tasks. This is equivalent to performing a round-robin assignment of the tasks to the nodes.

```
# 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
```

5.4.7.2.2 "autotuned" algorithm

This is an: adaptive heuristic algorithm based on a [simulated annealing](#) technique. "Adaptive" means that the number of tasks sent to each node varies, depending on the node's past performance and the nature of the workload.

```
# 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
```

5.4.7.2.3 “proportional” algorithm

This is an adaptive algorithm based on the contribution of each node to the overall mean task execution time.

```
# 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
```

5.4.7.2.4 “rl2” algorithm

This is an adaptive algorithm based on an artificial intelligence technique called “[reinforcement learning](#)”

```
# algorithm name
jppf.load.balancing.algorithm = rl2
# name of the set of parameter values or profile for the algorithm
jppf.load.balancing.profile = rl2_profile
# "rl2" profile
jppf.load.balancing.profile.rl2_profile.performanceCacheSize = 1000
jppf.load.balancing.profile.rl2_profile.performanceVariationThreshold = 0.75
jppf.load.balancing.profile.rl2_profile.minSamples = 20
jppf.load.balancing.profile.rl2_profile.maxSamples = 100
jppf.load.balancing.profile.rl2_profile.maxRelativeSize = 0.5
```

5.4.7.2.5 “nodethreads” algorithm

With this algorithm, each node will receive at most $n * m$ tasks, where n is the number of processing threads in the node and m is a user-defined parameter named 'multiplier'. Note that the number of processing threads of a node can be changed dynamically through the JPPF management features, in which case the algorithm will be notified and adapt accordingly.

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

5.4.7.3 Load-balancing documentation references

For a detailed explanation of the load-balancing in JPPF, its APIs and the predefined algorithms, please refer to the [Load Balancing](#) section of this manual.

Defining a custom algorithm is described in the “[creating a custom load-balancer](#)” section of this manual.

5.4.8 Maximum number of concurrent jobs per node

A node can process multiple jobs concurrently. The maximum number of concurrent jobs for all the nodes connected to a server is set with the following property:

```
# maximum number of jobs that can be sent to a node concurrently
jppf.node.max.jobs = 20
```

The actual value used for each node is determined as follows:

- when this property is unspecified in the node, it will default to the value of the property defined in the server
- when the property is defined neither in the node nor in the server, it will default to [Integer.MAX_VALUE](#), that is, $2^{31} - 1$ or 2147483647
- when this property is defined in both node and server, the value set in the node overrides the value set in the server

Note: the same override mechanism applies to connections between two servers

5.4.9 Configuring a local node

Each JPPF driver can 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 property, which defaults to "false":

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

Note 1: 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

Note 2: for the same reason, the SSL configuration does not apply to a local node

5.4.10 Heartbeat-based connection failure detection

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

- the JPPF node or client - designated here as heartbeat client - establishes a specific connection to the server, dedicated to failure detection
- at connection time, a handshake protocol takes place, where the heartbeat client communicates a unique id to the server, that can be correlated to other connections for this heartbeat client (job data channel, distributed class loader)
- at regular intervals (heartbeats), the server will send a very short message to the heartbeat client, who will acknowledge it by sending a short response of its own
- if the heartbeat client's response is not received in a specified time frame (heartbeat timeout), and this, a specified number of times in a row (heartbeat retries), the server will consider the connection to be broken, will close it cleanly, close the associated connections, and handle the recovery, such as requeuing tasks that were being executed
- on the heartbeat client side, if no message is received from the server for a time greater than `heartbeat_timeout * heartbeat_retries`, then it will close its connection to the server and attempt to reconnect.

In practice, the polling of the heartbeat clients 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 or clients.

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. Defaults to false (disabled).
jppf.recovery.enabled = true
# Maximum number of attempts to get a response from 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 15000 (15 seconds)
jppf.recovery.read.timeout = 15000
# Number of threads allocated to the reaper, defaults to the number of available CPUs
jppf.recovery.reaper.pool.size = 8
```

5.4.11 Redirecting the console output

In some situations, it might be desirable to redirect the standard and error output of the driver, that is, the output of `System.out` and `System.err`, to files. This can be accomplished with the following properties:

```
# file on the file system where System.out is redirected
jppf.redirect.out = /some/path/someFile.out.log
# whether to append to an existing file or to create a new one
jppf.redirect.out.append = false
# file on the file system where System.err is redirected
jppf.redirect.err = /some/path/someFile.err.log
# whether to append to an existing file or to create a new one
jppf.redirect.err.append = false
```

By default, a new file is created each time the driver is started, unless “jppf.redirect.out.append = true” or “jppf.redirect.err.append = true” are specified. If a file path is not specified, then the corresponding output is not redirected.

5.4.12 Resolution of the nodes IP addresses

You can switch on or off the DNS name resolution for the nodes connecting to this driver, with the following property:

```
# whether to resolve the nodes' ip addresses into host names
# defaults to true (resolve the addresses)
org.jppf.resolve.addresses = true
```


5.5 Node configuration

5.5.1 Server discovery

In a JPPF node, the JPPF server discovery mechanisms are all implemented as [node connection strategy plugins](#). JPPF provides several built-in such strategies, including the default strategy which uses the node configuration to find a JPPF driver to connect to. This mechanism allows both automatic discovery via UDP multicast, and manual configuration of the connection. These two features are described in the following sections.

5.5.1.1 Discovery through UDP multicast

By default, JPPF nodes are configured to automatically discover active servers on the network. As we have seen in the [server discovery](#) section of the JPPF driver configuration, 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 broadcasts from the server.

5.5.1.1.1 Enabling and disabling UDP multicast

This is done with the following property, which defaults to true (enabled):

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

5.5.1.1.2 Configuration of UDP multicast

The configuration is performed by defining a multicast group and port number, as in this example showing their default values:

```
# 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
```

Note: the values of these properties must be the same as those defined in the [server configuration](#).

5.5.1.1.3 Inclusion and exclusion patterns

The following properties define inclusion and exclusion patterns for IPv4 and IPv6 addresses. They provide a means of controlling whether to connect to a server, based on its IP address. Each of these patterns defines a list of comma- or semicolon- separated patterns. The IPv4 patterns can be expressed in either [CIDR](#) notation, or in a syntax defined in the Javadoc for the class [IPv4AddressPattern](#). Similarly, IPv6 patterns can be expressed in [CIDR](#) notation or in a syntax defined in [IPv6AddressPattern](#). This enables filtering out unwanted IP addresses: the discovery mechanism will only allow addresses that are included and not excluded.

```
# 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 =
```

Let's take for instance the following pattern specifications:

```
jppf.discovery.include.ipv4 = 192.168.1.
jppf.discovery.exclude.ipv4 = 192.168.1.128-
```

The equivalent patterns in CIDR notation would be:

```
jppf.discovery.include.ipv4 = 192.168.1.0/24
jppf.discovery.exclude.ipv4 = 192.168.1.128/25
```

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 192.168.1.128 ... 192.168.1.255. Thus, we actually defined a filter that only accepts addresses in the range 192.168.1.0 ... 192.168.1.127.

These 2 patterns can in fact be rewritten as a single inclusion pattern:

```
jppf.discovery.include.ipv4 = 192.168.1.-127
```

or, in CIDR notation:

```
jppf.discovery.include.ipv4 = 192.168.1.0/25
```

5.5.1.2 Manual connection configuration

If server discovery is disabled, network access to a server must be configured manually. To this effect, the node requires the address or host on which the JPPF 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 or 11143 for the port number, depending on whether secure connectivity is disabled or enabled, respectively).

5.5.1.3 Enabling secure connectivity

To enable secure connectivity via SSL/TLS for the configured connections, simply set the following:

```
# enable SSL/TLS over the discovered connections; defaults to false (disabled)
jppf.ssl.enabled = true
```

5.5.1.4 Heartbeat-based connection failure detection

The heartbeat mechanism to recover from hardware failure is enabled in the node with the following configuration property:

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

As described in the [server configuration counterpart](#), when the node hasn't received any heartbeat message for a time greater than `heartbeat_timeout * heartbeat_retries`, then it will close its connection to the server and attempt to reconnect.

5.5.1.5 Interaction between connection recovery and server discovery

When discovery is enabled for the node (`jppf.discovery.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

5.5.2 Node JVM options

In the same way as for a server (see [server JVM options](#)), the node is made of 2 processes: a "controller" process and a "node" process. The controller launches the node as a separate process and watches its exit code. If the exit code has a pre-defined value of 2, then the controller will restart the node process, otherwise it will simply terminate.

This mechanism allows the remote restart (eventually delayed) of a JPPF node 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 node process inherits the following parameters from the controller process:

- location of jppf configuration (`-Djppf.config` or `-Djppf.config.plugin`)
- location of Log4j configuration (`-Dlog4j.configuration`)
- 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 assertions enabled and remote debugging options:

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

Contrary to the Java command line, It is possible to specify multiple class path elements through this property, by adding one or more “-cp” or “-classpath” options. For example:

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

This syntax allows configuring multiple paths in an OS-independant way, in particular with regards to the path separator character (e.g. ':' on Linux, ';' on Windows).

If a classpath element contains one or more spaces, the path(s) it defines must be surrounded with double quotes:

```
jppf.jvm.options = -Xmx512m -cp "dir with spaces/myJar1.jar" -cp NoSpaces/myJar2.jar
```

5.5.3 Specifying the path to the JVM

It is possible to choose which JVM will run a node, by specifying the full path to the Java executable with the following property:

```
# Full path to the java executable
jppf.java.path = <path_to_java_executable>
# Linux example
jppf.java.path = /opt/jdk1.8.0/bin/java
# Windows example
jppf.java.path = C:/java/jdk1.7.0/bin/java.exe
```

This property is used in several situations:

- by the shell script from the node distribution that launches the node (startNode.sh or startNode.bat)
- by slave nodes when the property is specified as a configuration override, allowing to start a slave node with a different JVM than its master's
- when a node is restarted with one of the [JPPFNodeAdminMBean.updateConfiguration\(\)](#) management methods with `jppf.java.path` specified as an overridden property.

5.5.4 JMX management configuration

JPPF uses JMX to provide remote management capabilities for the nodes, and uses its own JMX 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 JPPF node runs its own JMX remote server. The port on which this JMX server will listen can be defined as follows:

```
# JMX management port; defaults to 11198
jppf.node.management.port = 11198
```

Note: if the specified JMX port is already in use, the JPPF node will lookup the next port numbers in sequence, until it finds one that is available.

5.5.5 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.

5.5.6 Maximum number of concurrent jobs

A node can process multiple jobs concurrently. The maximum number of concurrent jobs is set with the following property:

```
# maximum number of jobs the node can process concurrently
jppf.node.max.jobs = 20
```

The actual value used for the node is determined as follows:

- when this property is unspecified in the node, it will default to the value of the same property defined for the server to which the node is connected.
- when the property is defined neither in the node nor in the server, it will default to [Integer.MAX_VALUE](#), that is, $2^{31} - 1$ or 2147483647.
- when this property is defined in both node and server, the value set in the node overrides the value set in the server.

5.5.7 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.

5.5.8 Class loader resources cache

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

```
# whether the resource cache is enabled, defaults to 'true'
jppf.resource.cache.enabled = true
# 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 [local caching of network resources](#) section of this documentation.

5.5.9 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 on [offline nodes](#). To turn the offline mode on:

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

5.5.10 Redirecting the console output

As for JPPF drivers, the System.out and System.err of a node can be redirected to files, using the following properties:

```
# file on the file system where System.out is redirected
jppf.redirect.out = /some/path/someFile.out.log
# whether to append to an existing file or to create a new one
jppf.redirect.out.append = false
# file on the file system where System.err is redirected
jppf.redirect.err = /some/path/someFile.err.log
# whether to append to an existing file or to create a new one
jppf.redirect.err.append = false
```

By default, a new file is created each time the node is started, unless “jppf.redirect.out.append = true” or “jppf.redirect.err.append = true” are specified. If a file path is not specified, then the output is not redirected.

5.6 Client and administration console configuration

5.6.1 Server discovery in the client

Connection pools: when a JPPF client connects to one or more servers, its connections are organized by pools. All the connections in a pool share the same basic characteristics:

- name given to the pool, used as a prefix for individual connections names. A pool named "pool" will name its individual connections as "pool-1", ..., "pool-N", where N is the number of connections in the pool.
- server host or IP address
- server port
- whether secure connectivity via SSL/TLS is enabled
- pool priority: allows defining a failover hierarchy of connection pools, where the client only uses the available pool(s) with the highest priority. Whenever these pools become unavailable for any reason, the client will fall back to the pools with the next highest priority, switching back to the previous pools later on if they become available again
- associated JMX connections pool size: each connection pool also maintains one or more JMX connections to allow the administration and monitoring of the corresponding server

Pool size: in addition to the properties above, each connection pool has a *size*, which determines how many connections it manages. The size of a connection pool can be defined statically in the configuration, but it can also be changed programmatically using the [connection pools API](#).

Definition: *server discovery is the mechanism by which a JPPF client finds which servers to connect to, how to connect to them and organizes this information into a set of connections pools.*

In a JPPF client, all server discovery strategies are implemented as [discovery plugins](#). By default, JPPF provides a built-in discovery mechanism that uses the client configuration properties to find which servers to connect to. This mechanism allows automatic discovery via UDP multicast, along with the manual configuration of the connections pools. These two features are described in the following sections.

5.6.1.1 Discovery through UDP multicast

By default, JPPF clients are configured to automatically discover active servers on the network via [UDP multicast](#). With this mechanism, the server broadcasts data packets on the network that contain sufficient information to establish a standard TCP/IP connection.

5.6.1.1.1 Enabling and disabling UDP multicast

This is done with the following property, which defaults to true (enabled):

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

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.

5.6.1.1.2 Configuration of UDP multicast

The configuration is performed by defining a multicast group and port number, as in this example showing their default values:

```
# 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
```

Note: *the values of these properties must be the same as those defined in the server configuration.*

5.6.1.1.3 Connection pool size

The JPPF client will manage a pool of connections for each discovered server. 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
```

5.6.1.1.4 JMX Connection pool size

Each server connection pool has an associated pool of JMX connections, whose size is configured as follows:

```
# JMX connection pool size, defaults to 1
jppf.jmx.pool.size = 1
```

5.6.1.1.5 Jobs concurrency

Each connection in a pool can handle multiple jobs concurrently. The number of jobs each connection can handle is defined with the following property:

```
# Number of concurrent jobs each connection can handle. Defaults to Integer.MAX_VALUE
jppf.max.jobs = 100
```

By default, the maximum number of concurrent jobs is set to [Integer.MAX_VALUE](#), that is, $2^{31} - 1$, or 2,147,483,647.

5.6.1.1.6 Connections naming

Each server connection has an assigned name, following the pattern: "jppf_discovery-<n>-<p>", where n is a driver number, in order of discovery, and p is the connection number within the corresponding connection pool. For instance, if we defined `jppf.pool.size = 2`, then the first discovered driver will have 2 connections named "jppf_discovery-1-1" and "jppf_discovery-1-2"

5.6.1.1.7 Enabling secure connectivity

To enable secure connectivity via SSL/TLS for the discovered connections, simply set the following:

```
# enable SSL/TLS over the discovered connections
jppf.ssl.enabled = true
```

5.6.1.1.8 Connections pools priority

It is also possible to specify the priority of all discovered server connections, so that they will easily fit into a [failover or load-balancing strategy](#):

```
# priority assigned to all auto-discovered connections; defaults to 0
# this is equivalent to "<driver_name>.jppf.priority" in manual network configuration
jppf.discovery.priority = 10
```

5.6.1.1.9 Inclusion and exclusion patterns

The following four properties define inclusion and exclusion patterns for IPv4 and IPv6 addresses. They provide a means of controlling whether to connect to a server, based on its IP address. Each of these patterns defines a list of comma- or semicolon- separated patterns. The IPv4 patterns can be expressed in either [CIDR](#) notation, or in a syntax defined in the Javadoc for the class [IPv4AddressPattern](#). Similarly, IPv6 patterns can be expressed in [CIDR](#) notation or in a syntax defined in [IPv6AddressPattern](#). This enables filtering out unwanted IP addresses: the discovery mechanism will only allow addresses that are included and not excluded.

```
# 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 =
```

Let's take for instance the following pattern specifications:

```
jppf.discovery.include.ipv4 = 192.168.1.
jppf.discovery.exclude.ipv4 = 192.168.1.128-
```

The equivalent patterns in CIDR notation would be:

```
jppf.discovery.include.ipv4 = 192.168.1.0/24
jppf.discovery.exclude.ipv4 = 192.168.1.128/25
```


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 192.168.1.128 ... 192.168.1.255. Thus, we actually defined a filter that only accepts addresses in the range 192.168.1.0 ... 192.168.1.127.

These 2 patterns can in fact be rewritten as a single inclusion pattern:

```
jppf.discovery.include.ipv4 = 192.168.1.-127
```

or, in CIDR notation:

```
jppf.discovery.include.ipv4 = 192.168.1.0/25
```

5.6.1.1.10 Accepting multiple network interfaces per server

Additionally, you can specify 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 defaults to false, meaning that only the first discovered interface for a driver will be taken into account.

5.6.1.1.11 Heartbeat-based connection failure detection

To enable the detection of connection failure through a heartbeat mechanism, set the following property:

```
# enable the heartbeat mechanism for all discovered drivers; defaults to false (disabled)
jppf.recovery.enabled = true
```

Note 1: the heartbeat mechanism must also be enabled in the remote [server's configuration](#)
Note 2: this setting applies to all servers discovered via UDP multicast

5.6.1.2 Manual network configuration

As we have seen, a JPPF client can connect to multiple drivers. The first step will thus be to list and 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 attributes, each of them suffixed with "driver-1." or "driver-2.".

5.6.1.2.1 Connection to the JPPF server

The host name (or IP address) and port of each named server are defined as follows:

```
# 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
```

When left unspecified, they will default to "localhost" and "11111", respectively.

5.6.1.2.2 Connection pool size

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

Note that, contrary to UDP multicast discovery, each manually configured connection pool can have a different size

5.6.1.2.3 JMX Connection pool size

The size of the associated JMX connection pool is configured as follows:

```
# JMX connection pool size, defaults to 1
driver-1.jppf.jmx.pool.size = 1
```

5.6.1.2.4 Jobs concurrency

Each connection in a pool can handle multiple jobs concurrently. The number of jobs each connection can handle is defined with the following property:

```
# Number of concurrent jobs each connection can handle. Defaults to Integer.MAX_VALUE
driver-1.jppf.max.jobs = 100
```

By default, the maximum number of concurrent jobs is set to [Integer.MAX_VALUE](#), that is, $2^{31} - 1$, or 2,147,483,647.

5.6.1.2.5 Enabling secure connectivity

To enable secure connectivity via SSL/TLS for the configured connections, simply set the following:

```
# enable SSL/TLS over the discovered connections; defaults to false (disabled)
driver-1.jppf.ssl.enabled = true
```

5.6.1.2.6 Priority

```
# assigned driver priority; defaults to 0
driver-1.jppf.priority = 10
```

The priority assigned to a server connection enables the definition 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 will use connections with the next highest priority in the remaining accessible server connection pools.

5.6.1.2.7 Heartbeat-based connection failure detection

To enable the detection of connection failure through a heartbeat mechanism, set the following property:

```
# enable the heartbeat mechanism for all discovered drivers; defaults to false (disabled)
driver-1.jppf.recovery.enabled = true
```

Note: the heartbeat mechanism must also be enabled in the remote server's [server's configuration](#).

5.6.1.3 Using manual configuration and UDP multicast together

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

```
# enable discovery
jppf.discovery.enabled = true
# specify 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
```

5.6.2 Load-balancing and failover of server connection pools

Connection pools can be organized in any combination of two modes, using the priority attribute of each pool:

Load-balancing mode occurs when the connection pools have the same priority. In this case, the JPPF client will balance the jobs between the connections of the pools, according to the load-balancer settings. Example configuration:

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

driver-1.jppf.server.host = my.host1.com
driver-1.jppf.server.port = 11111
driver-1.jppf.priority = 20

driver-2.jppf.server.host = my.host2.com
driver-2.jppf.server.port = 11111
driver-2.jppf.priority = 20
```

Failover mode occurs when connection pools have different priorities. In this case, the JPPF client will only send jobs through the pool(s) with the highest priority. If the highest priority pool fails for any reason, the JPPF client will then fall back to the highest priority among the remaining connection pools, and so on. The following example shows such a configuration:

```
jppf.drivers = primary-pool secondary-pool

primary-pool.jppf.server.host = my.host1.com
primary-pool.jppf.server.port = 11111
primary-pool.jppf.priority = 20

secondary-pool.jppf.server.host = my.host2.com
secondary-pool.jppf.server.port = 11111
secondary-pool.jppf.priority = 10
```

You can also combine load-balancing and failover modes, since there is no limit to the number of connection pools you can define. For example:

```
jppf.drivers = primary-pool-1 primary-pool-2 secondary-pool-1 secondary-pool-2

primary-pool-1.jppf.server.host = my.host1.com
primary-pool-1.jppf.server.port = 11111
primary-pool-1.jppf.priority = 20

primary-pool-2.jppf.server.host = my.host2.com
primary-pool-2.jppf.server.port = 11111
primary-pool-2.jppf.priority = 20

secondary-pool-1.jppf.server.host = my.host3.com
secondary-pool-1.jppf.server.port = 11111
secondary-pool-1.jppf.priority = 10

secondary-pool-2.jppf.server.host = my.host4.com
secondary-pool-2.jppf.server.port = 11111
secondary-pool-2.jppf.priority = 10
```

5.6.3 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 multiple 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
```

A priority can be assigned to the local executor, so that it will easily fit into a failover strategy defined via the [manual network configuration](#):

```
# priority assigned to the local executor; defaults to 0
# this is equivalent to "<driver_name>.jppf.priority" in manual network configuration
jppf.local.execution.priority = 10
```

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 algorithm 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
```

Important note: when remote execution is disabled with `jppf.remote.execution.enabled = false`, neither [UDP multicast discovery](#) nor [manual network configuration](#) settings will have any effect anymore. In effect, this property allows to completely disable the built-in server discovery mechanisms, while still allowing optional [custom discovery mechanisms](#) to work.

5.6.4 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 = manual
# name of the set of parameter values (aka profile) to use for the algorithm
jppf.load.balancing.profile = jppf
# "jppf" profile
jppf.load.balancing.profile.jppf.size = 1000000
```

Also note that the load balancing is active even if only one remote execution is available. This has an impact on how tasks within a job will be sent to the 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.

5.6.5 Default execution policies

A JPPF client can have a default client and/or server side execution policy. The default policies are applied only to submitted jobs that don't have an execution policy. They can be set in the configuration with the following properties:

```
# server-side default execution policy
jppf.job.sla.default.policy = xml_source_type | xml_source
# client-side default execution policy
jppf.job.client.sla.default.policy = xml_source_type | xml_source
```

The `xml_source_type` part of the value specifies where to read the XML policy from, and the meaning of `xml_source` depends on its value. The value of `xml_source_type` can be one of:

- **inline**: `xml_source` is the actual XML policy specified inline in the configuration
- **file**: `xml_source` represents a path, in either the file system or classpath, to an XML file or resource. The path is looked up first in the file system, then in the classpath if it is not present in the file system
- **url**: `xml_source` represents a URL to an XML file, including but not limited to, http, https, ftp and file urls.

Note: `xml_source_type` can be omitted, in which case it defaults to "inline".

Here is an example specifying an inline policy:

```
# by default, jobs only execute on nodes with at least 4 CPUs
jppf.job.sla.default.policy = inline | <jppf:ExecutionPolicy> \
  <AtLeast> \
    <Property>availableProcessors</Property> \
    <Value>4</Value> \
  </AtLeast> \
</jppf:ExecutionPolicy>
```

The above XML execution policy is equivalent to this Java expression:

```
new AtLeast("availableProcessors", 4).toXML();
```

This can be used in a [scripted property value](#), which allows a much less cumbersome expression for the execution policy, as in this example also omitting the "inline" source type:

```
# server-side policy as an inline javascript expression
jppf.job.sla.default.policy = \
  ${ new org.jppf.node.policy.AtLeast("availableProcessors", 4).toXML(); }$
```

Other examples of XML execution policies taken from a file and a URL:

```
# default client-side policy from a file
jppf.job.client.sla.default.policy = file | ./config/defaultClientPolicy.xml
# default server-side policy from a URL
jppf.job.sla.default.policy = url | http://www.myhost.com/config/defaultServerPolicy.xml
```

5.6.6 Resolution of the drivers IP addresses

You can switch on or off the DNS name resolution for the drivers a client connects, with the following property:

```
# whether to resolve the drivers' ip addresses into host names
# defaults to true (resolve the addresses)
org.jppf.resolve.addresses = true
```

5.6.7 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.

5.6.8 UI refresh intervals in the administration tool

You may change the values of these properties if the graphical administration and monitoring tool is having trouble displaying all the information received from the nodes and servers. This may happen when the number of nodes and servers becomes large and the UI cannot cope. Increasing the refresh intervals (or decreasing the frequency of the updates) in the UI resolves such situations. The available configuration properties are defined as follows:

```
# refresh interval for the statistics panel in millis; defaults to 1000
# this is the interval between 2 successive stats requests to a driver via JMX
jppf.admin.refresh.interval.stats = 1000

# refresh interval in millis for the topology panels: tree view and graph view
# this is the interval between 2 successive runs of the task that refreshes the
# topology via JMX requests; defaults to 1000
jppf.admin.refresh.interval.topology = 1000

# refresh interval for the JVM health panel in millis; defaults to 1000
# this is the interval between 2 successive runs of the task that refreshes
# the JVM health via JMX requests
jppf.admin.refresh.interval.health = 1000

# UI refresh interval for the job data panel in ms. Its meaning depends on the
# publish mode specified with property "jppf.gui.publish.mode" (see below):
# - in "immediate_notifications" mode, this is not used
# - in "deferred_notifications" mode, this is the interval between 2 publications
#   of updates as job monitoring events
# - in "polling" mode this is the interval between 2 polls of each driver
jppf.gui.publish.period = 1000

# UI refresh mode for the job data panel. The possible values are:
# - polling: the job data is polled at regular intervals and updates to the view are
#   computed as the differences with the previous poll. This mode generates less network
#   traffic than the other modes, but some updates, possibly entire jobs, may be missed
# - deferred_notifications: updates are received as jmx notifications and published at
#   regular intervals, possibly aggregated in the interval. This mode provides a more
#   accurate view of the jobs life cycle, at the cost of increased network traffic
# - immediate_notifications: updates are received as jmx notifications and are all
#   published immediately as job monitoring events, which are pushed to the UI. In this
#   mode, no event is missed, however this causes higher cpu and memory consumption
# The default value is immediate_notifications
jppf.gui.publish.mode = immediate_notifications
```

5.6.9 Customizing the administration console's splash screen

At startup, the desktop administration console displays a splash screen made of a sequence of rolling images with a fixed text at the center. The splash screen can be customized with the following properties:

```
# Whether to display the animated splash screen at console startup, defaults to false
jppf.ui.splash = true
# The fixed text displayed at center of the window
jppf.ui.splash.message = The JPPF Admin Console is starting ...
# The message's font color, expressed as an rgb or rgba value. If alpha is not
# specified, it is assumed to be 255 (fully opaque).
# Examples: 255, 233, 127 (opaque) | 255, 233, 127, 128 (semi-transparent)
jppf.ui.splash.message.color = 64, 64, 128
# One or more paths to the images displayed in a rolling sequence (like a slide show)
# The images may be either in the file system or in the classpath and are separated with # '|'
# (pipe) characters
```

```
jppf.ui.splash.images = image_path_1 | ... | image_path_N  
# interval between images in milliseconds  
jppf.ui.splash.delay = 500
```

5.7 Common configuration properties

5.7.1 Lookup of classpath resources in the file system

By default, the JPPF distributed class loader looks for requested resources in the file system (using the resource name as file path), if they are not found in the class path. This behavior can be disabled by setting the following property in a node, server or client:

```
# Enable or disable the lookup of classpath resources in the file system
jppf.classloader.file.lookup = true
```

Since this property applies to the lookup of remote resources, the file system lookup will occur if and only if this property is true in the node configuration *and* in the server or client configuration. Also note that its value is true by default.

5.7.2 Socket connections recovery and failover

When the connection to a server is interrupted, the node, client or peer server 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 = 0
# 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 0 second delay, for 60 seconds and with connection attempts at 1 second intervals.

5.7.3 JMX requests timeout

In some very rare cases, a JMX request may be stuck because an error occurred in the remote driver or node, without breaking the JMX connection. A typical example is when an `OutOfMemoryError` occurs while the remote peer is executing the request. To avoid this, it is possible to set JMX request timeout with this configuration property:

```
# Timeout in millis for JMX requests. Defaults to Long.MAX_VALUE (2^63 - 1)
jppf.jmx.request.timeout = $script{ java.lang.Long.MAX_VALUE }
```

5.7.4 Global performance tuning parameters

These configuration properties affect the performance and throughput of I/O operations in JPPF. The values provided in the vanilla JPPF distribution (listed below) are known to offer a good performance in most situations and environments. These properties are defined as follows:

```
# Size of send and receive buffer for socket connections
# Defaults to 32768 and must be in range [1024, 1024*1024]
jppf.socket.buffer.size = 65536

# Size of temporary buffers (including direct buffers) used in I/O transfers
# Defaults to 32768 and must be in range [1024, 1024*1024]
jppf.temp.buffer.size = 12288

# Maximum size of temporary buffers pool (excluding direct buffers).# When this size
# is reached, new buffers are still created, but not released into the pool, so they
# can be quickly garbage-collected.
# The size of each buffer is defined with ${jppf.temp.buffer.size}
# Defaults to 10 and must be in range [1, 2048]
jppf.temp.buffer.pool.size = 200

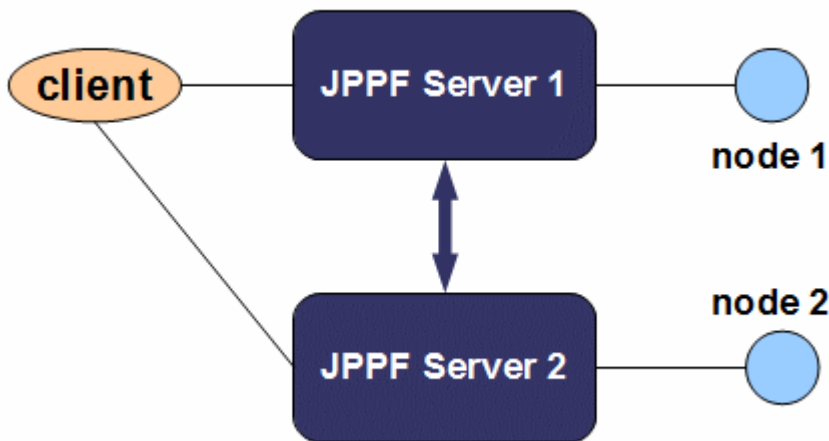
# Size of temporary buffer pool for reading lengths as ints (size of each buffer is 4)
# Defaults to 100 and must be in range [1, 2048]
jppf.length.buffer.pool.size = 100
```


5.8 Putting it all together

After seeing all the configuration details for the JPPF components, we would like to take a step back and have a look at the big picture of what the configuration is about: **defining a JPPF grid topology the way we want it**. For this purpose, we will define a specific non-trivial topology, and make two diagrams of this topology, where the JPPF components in each diagram are annotated with the minimal set of configuration properties that must be defined. One diagram presents the configuration when the connections between components are defined manually, the other diagram when they are automatically discovered.

5.8.1 Defining a topology

Here is the JPPF topology we want to achieve:

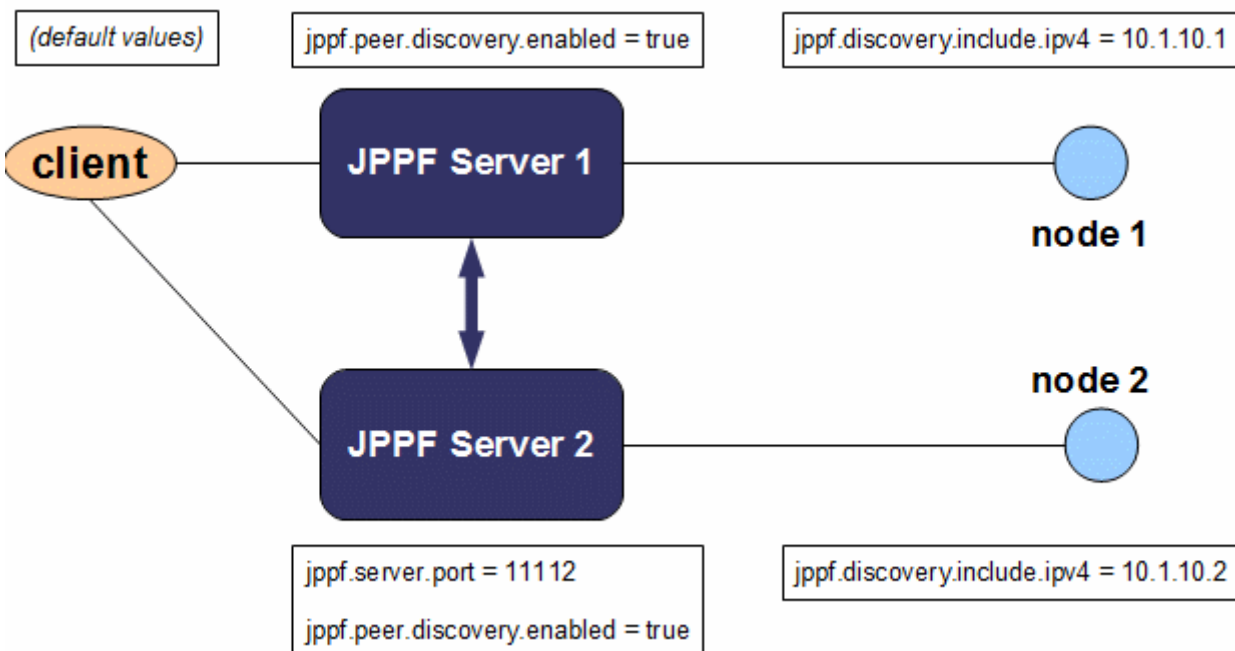


Let's state plainly the components involved and their relationships:

- there are one client, two servers (server1 and server2) and two nodes (node1 and node2)
- the client is connected to both servers
- each server is connected to the other server
- node1 is connected to server1
- node2 is connected to server2

In the following diagrams a property definition between parentheses means that the definition uses a default value, and thus doesn't need to be defined. This notation is used to emphasize what default values actually mean. For convenience, we will also assume that server1 is on a machine with IP address 10.1.10.1 and server2 on a machine with the IP address 10.1.10.2. They also listen on ports 11111 and 11112, respectively.

5.8.2 Automatic discovery of the topology

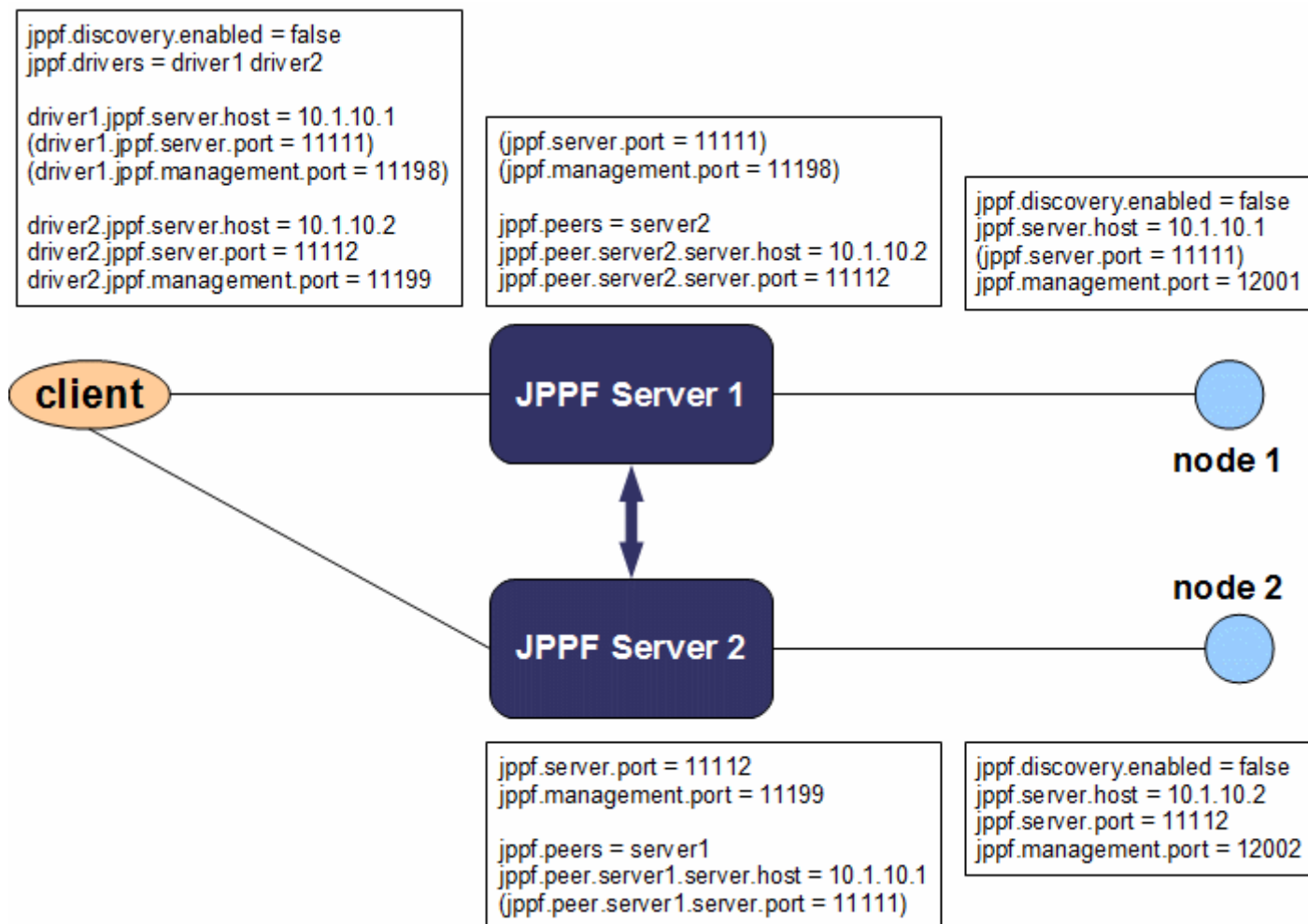


As we can see, automatic discovery involves very few configuration properties. The discovery itself is active by default (jppf.discovery.enabled = true). The client actually does not need any property at all and will work with an empty configuration. The drivers only need to override properties that define non-default values. Lastly, the nodes specify an

inclusive pattern for which server to connect to, otherwise they could pickup the wrong server, connect both to the same server, etc.

Also note that, if our topology only included server1, then no configuration property at all would be needed in the client, server1, node1 or node2. That is the power of automatic discovery in JPPF master/worker topologies.

5.8.3 Manual configuration of the topology



5.9 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.

5.9.1 Enabling secure connectivity

5.9.1.1 In the clients

There is a global property which applies to all drivers found via the [auto-discovery mechanism](#):

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

For drivers specified manually in the client configuration, SSL connectivity is specified individually:

```
# disable discovery
jppf.discovery.enabled = false
# manually specify two drivers
jppf.drivers = driver1 driver2

# define secure connectivity to driver1
driver1.jppf.server.host = secure_host
driver1.jppf.server.port = 11443
# secure connections are made to this driver
driver1.jppf.ssl.enabled = true

# define non-secure connectivity to driver2
driver2.jppf.server.host = non_secure_host
driver2.jppf.server.port = 11111
# the default value is false and needs not be specified
driver2.jppf.ssl.enabled = false
```

5.9.1.2 In the nodes

Nodes 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. This has an impact on the driver configuration: since the driver connects to the node's JMX server, it will be considered a SSL *client* from the JMX perspective. This means the driver will need to have the proper node certificate in its trust store.

5.9.1.3 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
```

5.9.2 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.

5.9.2.1 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")`.

5.9.2.2 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
```

5.9.3 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.

5.9.3.1 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](#).

5.9.3.2 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](#).

5.9.3.3 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](#).

5.9.3.4 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
```

5.9.3.5 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.keystore.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
```

Lastly, the format of the key store is specified with the following property:

```
# The key store format to use. When unspecified, it defaults to the value of
# KeyStore.getDefaultType(). For instance on the Oracle JVM it will be 'JKS',
# on Android's Dalvik it will be BKS, etc.
jppf.ssl.keystore.type = JKS
```

Note that, depending on the environment on which a node is executing, you may need to use different formats for the node's key store (if using mutual authentication) and for the driver's key store. For instance, if the node is running on Android, the "JKS" format, which is the default on the Oracle JVM, will not be supported and the key store will need to be converted to e.g. BKS (BouncyCastle KeyStore), which is the default on Android.

5.9.3.6 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 optional parameters
jppf.ssl.truststore.password.source = org.jppf.ssl.PlainTextPassword password
```

As for the key store, the format of the trust store is specified with the following property:

```
# the format to use for the trust store. When unspecified, it defaults to the
# value of KeyStore.getDefaultType(). For instance on the Oracle JVM it will be
# 'JKS', on Android's Dalvik it will be BKS, etc.
jppf.ssl.truststore.type = JKS
```

5.9.3.7 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 optional parameters
jppf.ssl.client.truststore.password.source = org.jppf.ssl.PlainTextPassword password

# the format to use for the client trust store. When unspecified, it defaults to
# the value of KeyStore.getDefaultType(). For instance on the Oracle JVM it will
# be 'JKS', on Android's Dalvik it will be BKS, etc.
jppf.ssl.client.truststore.type = JKS

```

In this configuration, a client will not be able to authenticate with the server if it uses a copy of the node's keystore, even if it uses the correct password.

5.9.4 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

# key store format
jppf.ssl.keystore.type = JKS
# 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

# trust store format
jppf.ssl.truststore.type = JKS
# 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

```


5.10 The JPPF configuration API

5.10.1 General properties handling

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, double, char and boolean, and also String and File.

Here is a summary of the API provided by TypedProperties:

```
public class TypedProperties extends AbstractTypedProperties {
    // 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)
    public TypedProperties setString(String key, String value)
    // int properties
    public int getInt(String key)
    public int getInt(String key, int defValue)
    public TypedProperties setInt(String key, int value)
    // long properties
    public long getLong(String key)
    public long getLong(String key, long defValue)
    public TypedProperties setLong(String key, long value)
    // float properties
    public float getFloat(String key)
    public float getFloat(String key, float defValue)
    public TypedProperties setFloat(String key, float value)
    // double properties
    public double getDouble(String key)
    public double getDouble(String key, double defValue)
    public TypedProperties setDouble(String key, double value)
    // char properties
    public char getChar(String key)
    public char getChar(String key, char defValue)
    public TypedProperties setChar(String key, char value)
    // boolean properties
    public boolean getBoolean(String key)
    public boolean getBoolean(String key, boolean defValue)
    public TypedProperties setBoolean(String key, boolean value)
    // File properties
    public File getFile(String key)
    public File getFile(String key, File defValue)
    public TypedProperties setFile(String key, File value)
}
```

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. Most of them also have a corresponding fluent setter method, which allows setting multiple properties of different types in a call chain.

It is possible to alter the JPPF configuration, via calls to the setXXX(String, XXX) methods of TypedProperties. 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:

```
// get the configuration and set the connection properties programmatically
JPPFConfiguration.getProperties().setBoolean("jppf.discovery.enabled", false)
    .setString("jppf.drivers", "driver1")
    .setString("driver1.jppf.server.host", "www.myhost.com")
    .setInt("driver1.jppf.server.port", 11111);
// now our configuration will be used
JPPFClient client = new JPPFClient();
```

5.10.2 Predefined JPPF properties

The JPPF properties documented in the [Configuration properties reference](#) chapter are also available as instances of the [JPPFProperty](#) interface, which is defined as follows:

```
public interface JPPFProperty<T> extends Serializable {
    // Get the name of this property
    String getName();

    // Get the default value of this property
    T getDefaultValue();

    // Get the aliases for this property, that is, other names it may be known as,
    // such as legacy names from prior versions
    String[] getAliases();

    // Convert the specified value into the type of values handled by this property
    T valueOf(String value);

    // Convert the specified value to a string
    String toString(T value);

    // Return the class object for the type of values of this property
    Class<T> valueType();

    // Return a concise localized description of the property
    String getDocumentation();
}
```

The class [JPPFProperties](#) holds a static enumeration of the predefined properties, each with a specific type parameter. For instance, the property whose name is "jppf.server.port" holds an int value and is obtained like this:

```
JPPFProperty<Integer> prop = JPPFProperties.SERVER_PORT;
System.out.printf("name: %s, default: %s, type: %s\n",
    prop.getName(), prop.getDefaultValue(), prop.valueType().getName());
```

As much as possible, the names of the constants in [JPPFProperties](#) are kept similar to the actual properties names: the constant names are all in upper case characters, the "jppf" prefix is removed, and the dots are replaced with underscores. For example "jppf.local.execution.enabled" will correspond to the LOCAL_EXECUTION_ENABLED constant.

The class [TypedProperties](#) has specific methods that use predefined properties:

```
public class TypedProperties extends Properties {
    // Get the value of a predefined property
    public <T> T get(JPPFProperty<T> property);

    // Set the value of a predefined property
    public <T> TypedProperties set(JPPFProperty<T> property, T value);

    // Remove the specified predefined property
    public <T> T remove(JPPFProperty<T> property);
}
```

Example usage:

```
int nbThreads = JPPFConfiguration.getProperties().get(JPPFProperties.PROCESSING_THREADS);
JPPFConfiguration.getProperties().set(JPPFProperties.DISCOVERY_ENABLED, true)
    .set(JPPFProperties.POOL_SIZE, 5).setString("custom.property", "some value");
```

Note the automatic type inference that is automatically performed by the compiler, as well as the preservation of the fluent interface for setting properties, allowing both custom and predefined properties to be set in the same statement.

Additionally, the `get(JPPFProperty)` method automatically returns the value of the property's `getDefaultValue()` method, whenever the property is not defined in the configuration.

[JPPFConfiguration](#) also has shortcut methods that make use of predefined properties:

```
public class JPPFConfiguration {
    // Get the value of a predefined property
    public static <T> T get(JPPFProperty<T> property);

    // Set the value of a predefined property
    public static <T> TypedProperties set(JPPFProperty<T> property, T value);

    // Remove the specified predefined property
    public static <T> T remove(JPPFProperty<T> property);
}
```

These methods resolve into equivalent calls to `JPPFConfiguration.getProperties().get/set/remove(...)`.

For example, these two statements do exactly the same thing

```
JPPFConfiguration.getProperties().set(JPPFProperties.SERVER_PORT, 11111);
JPPFConfiguration.set(JPPFProperties.SERVER_PORT, 11111);
```

Similarly for getting values:

```
int port;
port = JPPFConfiguration.getProperties().get(JPPFProperties.SERVER_PORT);
port = JPPFConfiguration.get(JPPFProperties.SERVER_PORT);
```

Combined with an `import static` statement, this leads to significantly smaller and more readable code:

```
import static org.jppf.utils.configuration.JPPFProperties.*;

...

int nbThreads = JPPFConfiguration.get(PROCESSING_THREADS);
JPPFConfiguration.set(DISCOVERY_ENABLED, true).set(POOL_SIZE, 5)
    .setString("custom.property", "some value").set(SERVER_PORT, 11111);
```

5.10.2.1 Parametrized properties

A parametrized property has one or more parts of its name defined as parameters which are substituted at runtime with actual string values. For instance, the following property is parameterized:

```
# define one RGB component of a named color
JPPFProperty<Integer> colorProp =
    new IntProperty("color.<color_name>.component.<rgb_comp>", 0):
```

Here, the parameters `color_name` and `rgb_comp` are recognized as such because they are enclosed within '`<`' and '`>`' delimiters. The class [TypedProperties](#) has overloads of the `get()`, `set()` and `remove()` methods to handle the parameters:

```
public class TypedProperties extends AbstractTypedProperties {
    // Get the value of a parametrized property
    public <T> T get(JPPFProperty<T> property, String...parameters);

    // Set the value of a predefined property
    public <T> TypedProperties set(JPPFProperty<T> property, T value, String...parameters);

    // Remove the specified predefined property
    public <T> T remove(JPPFProperty<T> property, String...parameters);
}
```

As an example, to fully define the color "cyan", we could write:

```
TypedProperties colors = new TypedProperties()
    .set(colorProp, 102, "cyan", "r") // red component
    .set(colorProp, 255, "cyan", "g") // green component
    .set(colorProp, 255, "cyan", "b"); // blue component
```

In a properties file, this would correspond to:

```
color.cyan.component.r = 102
color.cyan.component.g = 255
color.cyan.component.b = 255
```

Similarly, [JPPFConfiguration](#) overloads its `get()`, `set()` and `remove()` methods:

```
public class JPPFConfiguration {
    // Get the value of a parametrized property
    public static <T> T get(JPPFProperty<T> property, String...parameters);

    // Set the value of a parametrized property
    public static <T> TypedProperties set(
        JPPFProperty<T> property, T value, String...parameters);

    // Remove the specified parametrized property
    public static <T> T remove(JPPFProperty<T> property, String...parameters);
}
```

Lastly, the interface [JPPFProperty](#) also provides a `getParameters()` method which returns an array of the parameters names, which may be empty if the property is not parametrized..

The following example shows, using parametrized properties, how to [manually configure the connection from a client to a driver](#).

Using a configuration file:

```
# non-parametrized properties
jppf.discovery.enabled = false
jppf.drivers = driver1

# parametrized properties in the form <driver_name>.jppf.xxx
driver1.jppf.server.host = 192.168.1.24
driver1.jppf.server.port = 11111
driver1.jppf.priority = 20
driver1.jppf.pool.size = 3
```

By API:

```
String driver = "driver1";
JPPFConfiguration.set(JPPFProperties.DISCOVERY_ENABLED, false)
    .set(JPPFProperties.DRIVERS, new String[] { driver })
    .set(JPPFProperties.PARAM_SERVER_HOST, "192.168.1.24", driver)
    .set(JPPFProperties.PARAM_SERVER_PORT, 11111, driver)
    .set(JPPFProperties.PARAM_PRIORITY, 20, driver)
    .set(JPPFProperties.PARAM_POOL_SIZE, 3, driver);
```

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 they 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 immediately
    public void restart() throws Exception;
    // Restart the node once it is idle
    public void restart(Boolean interruptIfRunning) throws Exception;
    // Shutdown the node immediately.
    public void shutdown() throws Exception;
    // Shutdown the node once it is idle
    public void shutdown(Boolean interruptIfRunning) throws Exception;
    // Determine whether a deferred shutdown or restart was requested and not yet performed
    public NodePendingAction pendingAction();
    // Cancel a previous deferred shutdown or restart request, if any
    public void cancelPendingAction();
}
```

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

Calling `restart()` or `shutdown()` is equivalent to calling `restart(true)` or `shutdown(true)`, respectively. When any of these methods is invoked without the `interruptIfRunning` flag, or when the flag's value is `true`, the tasks that were being executed, if any, are automatically resubmitted to the server queue.

When the `interruptIfRunning` parameter is `false`, the node will wait until no more tasks are being executed before restarting or shutting down. Thus, `interruptIfRunning = false` indicates a deferred operation request.

It is possible to query whether a deferred action has been requested with the `pendingAction()` method, which returns a [PendingNodeAction](#) enum element, defined as follows:

```
public enum NodePendingAction {
    // There is no pending action
    NONE,
    // A deferred shutdown was requested
    SHUTDOWN,
    // A deferred restart was requested
    RESTART;
}
```

Finally, a deferred action can be cancelled with `cancelPendingAction()`, provided the action hasn't yet started.

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 and available storage space.
    JPPFSystemInformation systemInformation() throws Exception;
}
```

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

JPPFSystemInformation provides information about 6 different aspects of the environment:

```
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 JPPFNodeAdminMBean extends JPPFAdminMBean {
    // 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 JPPFNodeAdminMBean extends JPPFAdminMBean {
    // Update the configuration properties of the node and optionally restart the node.
    void updateConfiguration(Map<String, String> config, Boolean restart) throws Exception;
    void updateConfiguration(Map<String, String> config, Boolean restart,
        Boolean interruptIfRunning) throws Exception;
}
```

These methods send a set of configuration properties to the node, that will override those defined in the node's configuration file. The restart 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, or when you need to restart the node with a different JVM by specifying the `jppf.java.path` property.

Note that, when the restart parameter is true, the updateConfiguration() method will call restart() or restart(interruptIfRunning) after storing the configuration updates, depending on which method overload is invoked initially.

Finally, the interruptIfRunning parameter specifies whether the node should be restarted initially, or wait until it no longer has any task to execute.

6.1.2 Task-level monitoring

MBean name : **"org.jpff: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 in 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 successfully
    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();

    // Whether this is a user-defined notification sent from a task
    public boolean isUserNotification();
}
```

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()

    // Get the position of the task in the job to which it belongs
    public int getJobPosition()
}
```

[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.

Additionally, the method `isUserNotification()` allows you to unambiguously distinguish between user-defined notifications, sent via [Task.fireNotification\(Object, boolean\)](#), and task completion notifications automatically sent by the JPPF nodes.

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 `requestResourceCacheReset()` does not perform the reset immediately. 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 will be deleted, freeing space in the temporary files folder.

6.1.4 Node provisioning

Any JPPF node has the ability to start new nodes on the same physical or virtual machine, and stop and monitor these nodes afterwards. This provides a node provisioning facility, which allows to dynamically grow or shrink a JPPF grid based on the workload requirements.

This provisioning ability establishes a master/slave relationship between a standard node (master) and the nodes that it starts (slaves). Please note that a slave node cannot be in turn used as a master. Apart from this restriction, slave nodes can be managed and monitored as any other node.

6.1.4.1 Node provisioning MBean

Node provisioning is implemented with a dedicated Mbean, which exposes the [JPPFNodeProvisioningMBean](#) interface:

MBean name: **"org.jppf:name=provisioning,type=node"**

This is also the value of the constant [JPPFNodeProvisioningMBean.MBEAN_NAME](#).

[JPPFNodeProvisioningMBean](#) is defined as follows:

```
public interface JPPFNodeProvisioningMBean extends Serializable, NotificationEmitter {
    // The object name of this MBean
    String MBEAN_NAME = "org.jppf:name=provisioning,type=node";
    // Constant for notifications that a slave node has started
    String SLAVE_STARTED_NOTIFICATION_TYPE = "slave_started";
    // Constant for notifications that a slave node has stopped
    String SLAVE_STOPPED_NOTIFICATION_TYPE = "slave_stopped";

    // Get the number of slave nodes started by this MBean
    int getNbSlaves();

    // Start or stop the required number of slaves to reach the specified number,
    // with an interrupt flag set to true
    void provisionSlaveNodes(int nbNodes);
    // Same action, explicitly specifying the interrupt flag
    void provisionSlaveNodes(int nbNodes, boolean interruptIfRunning);
    // Start or stop the required number of slaves to reach the specified number,
    // using the specified configuration overrides and an interrupt flag set to true
    void provisionSlaveNodes(int nbNodes, TypedProperties configOverrides);
    // Same action, explicitly specifying the interrupt flag
    void provisionSlaveNodes(int nbNodes, boolean interruptIfRunning,
                             TypedProperties configOverrides);
}
```

The method `provisionSlaveNodes(int)` will start or stop a number of slave nodes, according to how many slaves are already started. For instance, if 4 slaves are already running and `provisionSlaveNodes(2)` is invoked, then 2 slaves will be stopped. Inversely, if no slave is running, then 2 slave nodes will be started.

The method `provisionSlaveNodes(int, TypedProperties)` behaves differently, unless the second argument is `null`. Since the argument of type [TypedProperties](#) specifies overrides of the slaves configuration, this means that all already running slaves must be restarted to take these configuration changes into account. Thus, this method will first stop all running slaves, then start the specified number of slaves, after applying the configuration overrides.

When the `interruptIfRunning` flag is set to false, it will cause slave nodes to stop only once they are idle, that is they will not stop immediately if they are executing tasks at the time the provisioning request is made. This flag is true by default, in the `provisionSlaveNodes()` methods that do not specify it. Therefore, the `provisionSlaveNodes(n)`, `provisionSlaveNodes(n, null)`, `provisionSlaveNodes(n, true)` and `provisioningSlaveNodes(n, true, null)` methods all have the same effect.

The following example shows how to start new slave nodes with a single processing thread each:

```
// connect to the node's JMX server
JMXNodeConnectionWrapper jmxNode = new JMXNodeConnectionWrapper(host, port, false);
// create a provisioning proxy instance
String mbeanName = JPPFNodeProvisioningMBean.MBEAN_NAME;
JPPFNodeProvisioningMBean provisioning = jmxNode.getProxy(
    mbeanName, JPPFNodeProvisioningMBean.class);
// set the configuration with a single processing thread
TypedProperties overrides = new TypedProperties().setInt("jppf.processing.threads", 1)
// start 2 slaves with the config overrides
provisioning.provisionSlaveNodes(2, overrides);
// check the number of slave nodes
int nbSlaves = provisioning.getNbSlaves();
// or, using jmxNode directly get it as a JMX attribute
nbSlaves = (Integer) jmxNode.getAttribute(mbeanName, "NbSlaves");
```

6.1.4.2 Provisioning notifications

The [node provisioning MBean](#) also emits notifications when a slave node is started or stopped. These notifications are standard JMX [Notification](#) instances, with the following characteristics:

- the notification type, obtained with [Notification.getType\(\)](#) is one of these constants in [JPPFNodeProvisioningMBean](#):
 - [SLAVE_STARTED_NOTIFICATION_TYPE](#)
 - [SLAVE_STOPPED_NOTIFICATION_TYPE](#)
- the notification's user data, obtained with [Notification.getUserData\(\)](#), is a [JPPFProvisioningInfo](#) object, defined as:

```
public class JPPFProvisioningInfo implements Serializable {
    // Get the uuid of the master node that launched the slave
    public String getMasterUuid()
    // Get the id of the slave, relevant only to the master
    public int getSlaveId()
    // Get the slave node process exit code
    public int getExitCode()
    // Get the command line used to start the slave node process
    public List<String> getLaunchCommand()
}
```

Note that `getExitCode()` always returns -1 for slave *started* notifications.

In the following example, we define a [NotificationListener](#) which receives provisioning notifications and prints out the relevant information for each slave node that is started or stopped:

```
public class MyProvisioningListener implements NotificationListener {
    @Override
    public void handleNotification(Notification notif, Object handback) {
        switch(notif.getType()) {
            // case when a slave node is started
            case JPPFNodeProvisioningMBean.SLAVE_STARTED_NOTIFICATION_TYPE:
                System.out.println("slave node started: " + notif.getUserData());
                break;

            // case when a slave node is stopped
            case JPPFNodeProvisioningMBean.SLAVE_STOPPED_NOTIFICATION_TYPE:
                System.out.println("slave node stopped: " + notif.getUserData());
                break;
        }
    }
}
```

The notification listener can then be used as follows:

```
try (JMXNodeConnectionWrapper nodeJmx = new JMXNodeConnectionWrapper("localhost", 12001)) {
    if (nodeJmx.connectAndWait(5000L)) {
        JPPFNodeProvisioningMBean provisioning = nodeJmx.getJPPFNodeProvisioningProxy();
        // register a provisioning notification listener
        provisioning.addNotificationListener(new MyProvisioningListener(), null, null);
        // start 2 slave nodes and wait for the output of the notification listener
        provisioning.provisionSlaveNodes(2);
    }
}
```

6.1.5 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.

6.1.5.1 Connecting to an MBean server

Connecting 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, port and secure flag:

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

Here host and port represent the node's configuration properties "jppf.management.host" and "jppf.management.port", and secure is a boolean flag indicating whether network transport is secured via SSL/TLS.

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

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 ...;
```

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

6.1.5.3 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, secure);
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);
```

6.1.5.4 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, secure);
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();
```

6.1.5.5 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, secure);
wrapper.connectAndWait(timeout);
JPPFNodeTaskMonitorMBean proxy = wrapper.getJPPFNodeTaskMonitorProxy();
// subscribe to all notifications from the MBean
proxy.addNotificationListener(myNotificationListener, null, null);
```

b. Using the MBeanServerConnection API

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

6.1.6 Remote logging

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

To configure Log4j to emit JMX notifications, edit the log4j configuration file 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 of the node and add:

```
# list of handlers
handlers= java.util.logging.FileHandler, org.jppf.logging.jdk.JmxHandler
# Write log messages as JMX notifications.
org.jppf.logging.jdk.JmxHandler.level = FINEST
org.jppf.logging.jdk.JmxHandler.formatter = org.jppf.logging.jdk.JPPFLogFormatter
```

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

```
// get a JMX connection to the node MBean server
JMXNodeConnectionWrapper jmxNode = new JMXNodeConnectionWrapper(host, port, secure);
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();
// subscribe 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 `JPPFDriverAdminMBean.MBEAN_NAME`.

This MBean's role is to perform management and monitoring of the server. It exposes the `JPPFDriverAdminMBean` 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 `JPPFStatistics`. 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 to the 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;
    // Request the JMX connection information for all nodes which satisfy the node selector
    public Collection<JPPFManagementInfo> nodesInformation(NodeSelector selector)
        throws Exception;

    // Get the number of nodes currently attached to the server.
    public Integer nbNodes() throws Exception;
    // Get the number of nodes attached to the driver that satisfy the specified selector
    public Integer nbNodes(NodeSelector selector) throws Exception;
}
```

Note that the methods using a [node selector](#) as input parameter provide a lot of flexibility in choosing which nodes to get information about.

The [JPPFManagementInfo](#) objects returned in the resulting collection encapsulate enough information to connect to the corresponding node's MBean server:

```
public class JPPFManagementInfo implements Serializable, Comparable<JPPFManagementInfo> {
    // Get the host on which the driver or node is running
    public synchronized String getHost();
    // Get the ip address of the host on which the node or driver is running
    public String getIpAddress();
    // 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();
    // Whether this information represents a master node for provisioning
    public boolean isMasterNode();
    // Whether this information represents a slave node for provisioning
    public boolean isSlaveNode() {
        // Determine whether the node is active or inactive
        public boolean getMasterUuid();
    }
    // Whether this information represents a node than can execute .Net tasks
    public boolean isDotnetCapable();
    // Whether this information represents an Android node
    public boolean isAndroidNode();
    // 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 (JPPFManagementInfo info: nodesInfo) {
    // create a JMX connection wrapper based on the node information
    JMXNodeConnectionWrapper wrapper =
        new JMXNodeConnectionWrapper(info.getHost(), info.getPort(), false);
    // connect to the node's MBean server
    wrapper.connectAndWait(5000L);
    // 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 information on the idle nodes attached to the server
    public Collection<JPPFManagementInfo> idleNodesInformation() throws Exception;
    // Request information on all idle nodes which satisfy the node selector
    public Collection<JPPFManagementInfo> idleNodesInformation(NodeSelector selector)
        throws Exception;
    // Get the number of idle nodes attached to the server
    public Integer nbIdleNodes() throws Exception;
    // Get the number of idle nodes that satisfy the specified selector
    public Integer nbIdleNodes(NodeSelector selector) 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 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 methods:

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
    // Toggle the activate state of the specified nodes
    public void toggleActiveState(NodeSelector selector) throws Exception;
    // Get the active states of the nodes specified with a NodeSelector
    Map<String, Boolean> getActiveState(NodeSelector selector) throws Exception;
    // Set the active state of the specified nodes
    void setActiveState(NodeSelector selector, boolean active) throws Exception;
}
```

The `toggleActiveState()` 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 specify 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 Driver UDP broadcasting state

The driver management MBean has a read-write attribute which allows monitoring and setting its ability to broadcast its connection information to clients, nodes or other servers, via UDP. This attribute is defined via the following accessors:

```
public interface JPPFDriverAdminMBean extends JPPFAdminMBean {
    // Determine whether the driver is broadcasting
    Boolean isBroadcasting() throws Exception;

    // Activate or deactivate the broadcasting of the driver's connection information
    void setBroadcasting(Boolean broadcasting) 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 jobId) throws Exception;
    // Cancel the selected jobs
    public void cancelJobs(JobSelector selector) throws Exception;
    // Suspend the job with the specified id
    public void suspendJob(String jobId, Boolean requeue) throws Exception;
    // Suspend the selected jobs
    public void suspendJobs(JobSelector selector, Boolean requeue) throws Exception;
    // Resume the job with the specified id
    public void resumeJob(String jobId) throws Exception;
    // Resume the selected jobs
    public void resumeJobs(JobSelector selector) throws Exception;
    // Update the maximum number of nodes a job can run on
    public void updateMaxNodes(String jobId, Integer maxNodes) throws Exception;
    // Update the maximum number of nodes for the selected jobs
    public void updateMaxNodes(JobSelector selector, Integer maxNodes) throws Exception;
    // Update the priority of a job
    void updatePriority(String jobId, Integer newPriority);
    // Update the priority the selected jobs
    void updatePriority(JobSelector selector, Integer newPriority);
    // Get the set of uids of all the jobs currently queued or executing
    public String[] getAllJobUids() throws Exception;
    // Get an object describing the job with the specified uid
    public JobInformation getJobInformation(String jobId) throws Exception;
    // Get information on the selected jobs
    public JobInformation[] getJobInformation(JobSelector selector) throws Exception;
    // Get a list of objects describing the nodes to which the whole
    // or part of a job was dispatched
    public NodeJobInformation[] getNodeInformation(String jobId) throws Exception;
    // Get the list of dispatches for each of the selected nodes
    public Map<String, NodeJobInformation[]> getNodeInformation(JobSelector selector)
        throws Exception;
}
```

Reminder:

A job can be made of multiple tasks. These tasks may not be all executed on the same node. Instead, the set of tasks may be split in several subsets, and these subsets can in turn be 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 Job selectors

Many of the methods in [DriverJobManagementMBean](#) use a [JobSelector](#), which allows filtering the jobs in the driver's queue according to user-defined criteria. Any job monitoring function or management request will only apply to the jobs selected by the provided selector. A job selector is always evaluated against the most recent state of the driver's job queue, therefore the set of jobs selected by a job selector may vary over time.

Job selectors are described in full details in a [dedicated section of this documentation](#).

6.2.2.2 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 uuid  
    public void cancelJob(String jobId) throws Exception;  
    // Cancel the selected jobs  
    public void cancelJobs(JobSelector selector) throws Exception;  
}
```

This will terminate the specified jobs. 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, it 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 jobId, Boolean requeue) throws Exception;  
    // Suspend the selected jobs  
    public void suspendJobs(JobSelector selector, Boolean requeue) throws Exception;  
}
```

These methods will suspend the specified jobs. 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 jobId) throws Exception;  
  
    // Resume the selected jobs  
    public void resumeJobs(JobSelector selector) throws Exception;  
}
```

These methods resume the execution of the specified jobs. If a job was not suspended, then the method has no effect.

6.2.2.3 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 jobId, Integer maxNodes) throws Exception;  
  
    // Update the maximum number of nodes for the selected jobs  
    public void updateMaxNodes(JobSelector selector, Integer maxNodes) throws Exception;  
}
```

These methods specify the maximum number of nodes the specified jobs 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 jobs.

6.2.2.4 Updating the priority of a job

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

These methods dynamically update the priority of the specified jobs. The update takes effect immediately.

6.2.2.5 Updating the job SLA and metadata

The SLA and metadata of one or more jobs can be updated dynamically, together, atomically and as a whole. This is done with the following method:

```
public interface DriverJobManagementMBean extends NotificationEmitter {  
    // Update the SLA and/or metadata of the specified jobs
```

```
void updateJobs(JobSelector selector, JobSLA sla, JobMetadata metadata);
}
```

Note that, if any or both of the `sla` and `metadata` parameters are null, they will be ignored. To perform the update on a single job, use a `JobUuidSelector` as follows:

```
JPPFJob job = ...;
JobSelector selector = new JobUuidSelector(job.getUuid());
```

Below is an example using this feature:

```
// get a proxy to the job management MBean
JMXDriverConnectionWrapper jmx = ...;
DriverJobManagementMBean jobManager = jmx.getJobManager();

// create a non-blocking job
JPPFJob job = new JPPFJob();
job.setBlocking(false);
for (int i=0; i<3; i++) job.add(new MyTask());
JobSLA sla = job.getSLA();
// disable execution on master nodes
sla.setExecutionPolicy(new Equal("jppf.node.provisioning.master", false));

// submit the job and do something else
client.submitAsync(job);
...

// remove the execution policy to allow execution on all nodes
sla.setExecutionPolicy(null);
// now update the SLA for our job
jobManager.updateJobs(new JobUuidSelector(job.getUuid()), sla, null);
...
```

6.2.2.6 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 the 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[] getAllJobUuids() throws Exception;
    // Get an object describing the job with the specified uuid
    public JobInformation getJobInformation(String jobUuid) throws Exception;
    // Get information on the selected jobs
    public JobInformation[] getJobInformation(JobSelector selector) 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;
    // Get the list of dispatches for each of the selected nodes
    public Map<String, NodeJobInformation[]> getNodeInformation(JobSelector selector)
        throws Exception;
}
```

The `getAllJobUuids()` 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 `getJobInformation(...)` methods retrieve information about the state of the specified job(s) in the server. These methods return one or more objects of type [JobInformation](#), 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 `getNodeInformation(...)` methods also allow to obtain information about all the sub-jobs of a job that are dispatched to remote nodes. The return value is an array of objects of type [NodeJobInformation](#), or a map of job uuid to such an array if using a selector, defined as follows:

```

public class NodeJobInformation implements Serializable {
    // Get the information about the node.
    public JPPFManagementInfo getNodeInfo()

    // Get the information about the job dispatch
    public JobInformation getJobInformation()
}

```

This class is simply a grouping of two objects of type [JobInformation](#) and [JPPFManagementInfo](#), which we have already seen previously. The `nodeInfo` attribute will allow us to connect to the corresponding node's MBean server and obtain additional job monitoring data.

Please also note that the method `getNodeInformation(JobSelector)` returns a mapping of job uuids to their corresponding node dispatches.

6.2.2.7 Job notifications

Whenever a job-related event occurs, the job management MBean will emit a notification of type [JobNotification](#), defined as follows:

```

public class JobNotification extends Notification {
    // the information about the job or sub-job
    public JobInformation getJobInformation()

    // the information about the node (for sub-jobs only)
    // null for a job on the server side
    public JPPFManagementInfo getNodeInfo()

    // the creation timestamp for this event
    public long getTimestamp()

    // the type of this job event
    public JobEventType getEventType()

    // Get the uuid of the driver which emitted the notification
    public String getDriverUuid()
}

```

The value of the job event type (see [JobEventType](#) type safe enumeration) is one of the following:

- `JOB_QUEUED`: a new job was submitted to the JPPF driver queue
- `JOB_ENDED`: a job was completed and sent back to the client
- `JOB_DISPATCHED`: a sub-job was dispatched to a node
- `JOB_RETURNED`: a sub job returned from a node
- `JOB_UPDATED`: one of the job attributes has changed

6.2.3 Accessing and using the server MBeans

As for the nodes, JPPF provides an API that simplifies access to the JMX-based management features of a server, by abstracting most of the complexity of JMX programming. This API is implemented by the wrapper 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 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 specifying the management host, port and secure flag:

```
JMXDriverConnectionWrapper wrapper = new JMXDriverConnectionWrapper(host, port, secure);
```

Here host and port represent the server's configuration properties "jppf.management.host" and "jppf.management.port", and secure is a boolean flag indicating whether the network transport is secured via SSL/TLS.

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, secure);
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, secure);
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();
```

JMXDriverConnectionWrapper also has a more convenient method `getJobManager()` to obtain a proxy to the job management MBean:

```
JMXDriverConnectionWrapper wrapper = ...;
// create the proxy instance
DriverJobManagementMBean proxy = wrapper.getJobManager();
// get the ids of all jobs in the server queue
String[] jobUuids = proxy.getAllJobUuids();
```

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, secure);
wrapper.connectAndWait(timeout);
DriverJobManagementMBean proxy = wrapper.getJobManager();
// subscribe to all notifications from the MBean
proxy.addNotificationListener(myJobNotificationListener, null, null);
```

b. Using the MBeanServerConnection API

```
JMXDriverConnectionWrapper wrapper = new JMXDriverConnectionWrapper(host, port, secure);
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 [NotificationListener](#) interface:

```
// this class prints a message each time a job is added to the server's queue
public class MyJobNotificationListener implements NotificationListener {
    // Handle an MBean notification
    public void handleNotification(Notification notification, Object handback) {
        JobNotification jobNotif = (JobNotification) notification;
        JobEventType eventType = jobNotif.getEventType();
        // print a message for new jobs only
        if (eventType == JobEventType.JOB_QUEUED) {
            String uuid = jobNotif.getJobInformation().getJobUuid();
            System.out.println("job " + uuid + " was queued at " + 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 driver 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
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, secure);
```

```
jmxDriver.connectAndWait(5000L);
// get a proxy to the MBean
JmxLogger loggerProxy = 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();
// subscribe to all notifications from the MBean
NotificationListener listener = new MyLoggingHandler();
loggerProxy.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();
        System.out.println(handback.toString() + ": " + message);
    }
}
```

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 The JPPFNodeForwardingMBean interface

All forwarding operations are performed by a special MBean in the driver, exposing the [JPPFNodeForwardingMBean](#) interface. The object name of this mbean is "**org.jppf:name=provisioning,type=node**", which is also the value of the constant [JPPFNodeForwardingMBean.MBEAN_NAME](#).

As we can see in the Javadoc, all the methods in this interface return a `Map<String, Object>`, where:

- the key represents the UUID of a node to which the request was forwarded
- the value is either the return value of the forwarded request for this specific node, or an `Exception` that was raised when executing the request for the node

Note: the values in the map may be null, indicating that the target MBean method in the node does not return a value (return type void) or that the return value was just null.

We can also see that all methods of [JPPFNodeForwardingMBean](#) take a [NodeSelector](#) as parameter, specifying the nodes to which the request is forwarded. The methods `forwardInvoke()`, `forwardGetAttribute()` and `forwardSetAttribute()` are general methods applied to a specified MBean of each selected node. The other methods apply to predefined MBeans registered in the node.

For instance, you could get the states of selected nodes in two different ways, as follows:

```
NodeSelector selector = NodeSelector.ALL_NODES;
JPPFNodeForwardingMBean forwarder = ...;
Map<String, Object> result = new HashMap<>();
// the state of the nodes can be obtained generically by calling forwardInvoke()
result = forwarder.forwardInvoke(selector, JPPFNodeAdminMBean.MBEAN_NAME, "state");
// the state can also be obtained by calling state() without need for an mbean name
result = forwarder.state(selector);
```

6.3.2 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:

```
// Marker interface for selecting nodes
public interface NodeSelector extends Serializable {
    // Constant for a selector which accepts all nodes
    NodeSelector ALL_NODES = new AllNodesSelector();
    // Determine whether the node is accepted
    boolean accepts(JPPFManagementInfo nodeInfo);
}
```

The participation of a node is determined by the `accepts()` method, which takes a [JPPFManagementInfo](#) argument that represents information on the node as seen by the server. In particular, it provides access to the same node-specific [JPPFSystemInformation](#) object that is used by execution policies.

6.3.2.1 Selecting all the nodes

[AllNodesSelector](#) will select all the nodes currently attached to the server. Rather than creating instances of this class, you can also use the predefined constant [NodeSelector.ALL_NODES](#). This class is defined as:

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

Example usage:

```
// these two selectors are equivalent
NodeSelector selector1 = new AllNodesSelector();
NodeSelector selector2 = NodeSelector.ALL_NODES;
```

6.3.2.2 Execution policy selector

ExecutionPolicySelector uses an execution policy, such as can be set upon a JPPF job's SLA, to perform the selection of the nodes. It is defined as follows

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

Example usage:

```
// define a selector that selects all nodes with at least 4 cores
ExecutionPolicy policy = new AtLeast("availableProcessors", 4);
NodeSelector selector = new ExecutionPolicySelector(policy);
```

6.3.2.3 UUID-based selector

UuidSelector will only select nodes whose UUID is part of the collection or array of specified UUIDs. It is defined as:

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

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 topology will be automatically taken into account by the selection mechanism.

Example usage:

```
// define a selector that selects all idle nodes at a given time
JMXDriverConnectionWrapper driver = ...;
// get the uuids of all idle nodes
Collection<JPPFManagementInfo> infos = driver.idleNodesInformation();
List<String> uuids = new ArrayList<>();
if (infos != null) {
    for (JPPFManagementInfo info: infos)
        uuids.add(info.getUuid());
}
NodeSelector selector = new UuidSelector(uuids);
```

6.3.2.4 Scripted node selector

A **ScriptedNodeSelector** uses a script expression that evaluates to a boolean to determine which nodes are accepted. The script language must be accepted by the JVM and must conform to the [JSR-223](#) / [javax.script](#) API specifications.

A scripted node selector is particularly useful when you need to implement a complex filtering logic but do not want to deploy the associated code in the driver's classpath.

ScriptedNodeSelector extends **BaseScriptEvaluator** and is defined as follows:

```
public class ScriptedNodeSelector extends BaseScriptEvaluator implements NodeSelector {
```

```

// Initialize this selector with the specified language and script
public ScriptedNodeSelector(String language, String script)
// Initialize this selector with the specified language and script reader
public ScriptedNodeSelector(String language, Reader scriptReader) throws IOException
// Initialize this selector with the specified language and script file
public ScriptedNodeSelector(String language, File scriptFile) throws IOException
}

```

The script will be evaluated for each [JPPFManagementInfo](#) passed as input to the `accepts()` method of the selector. The node information can be accessed from the script using a predefined variable name "nodeInfo", as in this example:

```

JPPFDriverAdminMBean driverAdmin = ...;
NodeSelector selector;
// create a selector with a Javascript script which accepts only master nodes
selector = new ScriptedNodeSelector("javascript", "nodeInfo.isMasterNode();");
// or an equivalent Groovy script
selector = new ScriptedNodeSelector("groovy", "return nodeInfo.isMasterNode()");
// get the number of selected nodes
int nbMasterNodes = driverAdmin.nbNodes(selector);

```

6.3.2.5 Custom node selector

When a node selector requires a complex logic and its performance is critical, you can always write your own implementation of [NodeSelector](#), as in this example:

```

public class CustomNodeSelector implements NodeSelector {
    @Override
    public boolean accepts(JPPFManagementInfo nodeInfo) {
        // retrieve the node's number of cores
        int n = nodeInfo.getSystemInformation().getRuntime().getInt("availableProcessors");
        // accept only slave nodes with at least 4 cores
        return nodeInfo.isSlaveNode() && (n >= 4);
    }
}

```

Note: since the node selector is evaluated by the driver, the code of its implementation, along with all the classes it depends on, must be deployed in the driver's classpath.

6.3.3 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 = ...;
JMXDriverConnectionWrapper driverJmx = client.awaitWorkingConnectionPool()
    .awaitWorkingJMXConnection();
// get a proxy to the node forwarding MBean
JPPFNodeForwardingMBean proxy = driverJmx.getProxy(
    JPPFNodeForwardingMBean.MBEAN_NAME, JPPFNodeForwardingMBean.class);
// or, in a much less cumbersome way:
proxy = driverJmx.getNodeForwarder();

// 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 ...
}
}

```

6.3.4 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 = ...;
JMXDriverConnectionWrapper driverJmx = client.awaitWorkingConnectionPool()
    .awaitWorkingJMXConnection();

// this selector selects all nodes attached to the driver
NodeSelector selector = NodeSelector.ALL_NODES;

// 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 abilities 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:

```
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:

```
// 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 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.printThreadDump(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

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:

```
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 $\text{SUM}_i\{\text{cpuTime}(\text{thread}_i)\} / \text{interval}$, 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 generated 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 calling [DiagnosticsMBean.gc\(\)](#). This method simply calls [System.gc\(\)](#) and thus has the exact same semantics and constraints.

7 Extending and Customizing JPPF

7.1 Global extensions

7.1.1 Alternate 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.

7.1.1.1 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();
    }
}
```

7.1.1.2 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](#).

7.1.1.3 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](#).

7.1.1.4 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 was thoroughly optimized, to the point where it is now faster than the default Java serialization, with a smaller serialized object size in most cases.

To specify this scheme in your JPPF configuration:

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

7.1.1.5 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

7.1.1.6 Kryo serialization sample

The [Kryo serialization sample](#) provides an implementation of `JPPFSerialization` which uses the [Kryo](#) library for serializing and deserializing Java objects.

7.1.2 Composite serialization

It is also possible to use a special type of serialization scheme that will be applied on top of the serialization scheme defined in the [previous section](#). This allows you to perform any kind of transformation to the data provided by the serialization, such as compression / decompression or encryption / decryption.

To implement such a transformation, you will need to create a class which extends the abstract class [JPPFCompositeSerialization](#), defined as follows:

```
public abstract class JPPFCompositeSerialization implements JPPFSerialization {
    // Get the concrete serialization to delegate to
    public final JPPFSerialization getDelegate()

    // Get the case-insensitive unique name given to this composite serialization
    public abstract String getName();
}
```

This design makes it is possible to define a chain of transformations on top of a concrete [JPPFSerialization](#). To this effect, the `getDelegate()` method provides a reference to the next transformation or serialization in the chain.

The `getName()` method assigns a unique, case-insensitive name to the defined transformation. If two transformations have the same name, the last one will override the first, which will then be unavailable.

For instance, the predefined [ZLIBSerialization](#) is implemented like this:

```
public class ZLIBSerialization extends JPPFCompositeSerialization {
    @Override
    public void serialize(Object o, OutputStream os) throws Exception {
        Deflater deflater = new Deflater();
        DeflaterOutputStream zlibos = new DeflaterOutputStream(os, deflater);
        try {
            // delegate to the next (composite) serialization in the chain
            getDelegate().serialize(o, zlibos);
        } finally {
            zlibos.finish();
            deflater.end(); // required to clear the native/JNI buffers
        }
    }

    @Override
    public Object deserialize(InputStream is) throws Exception {
        Inflater inflater = new Inflater();
        InflaterInputStream zlibis = new InflaterInputStream(is, inflater);
        try {
            // delegate to the next (composite) deserialization in the chain
            return getDelegate().deserialize(zlibis);
        } finally {
            inflater.end(); // required to clear the native/JNI buffers
        }
    }

    @Override
    public String getName() { return "ZLIB"; }
}
```

Composite serializations are made available to JPPF via the service provider interface (SPI):

- in your source folder create the file **META-INF/services/org.jppf.serialization.JPPFCompositeSerialization**
- in this file add, for each of your composite serialization implementations, its fully qualified class name, for instance: `com.my.company.MyCompositeSerialization`
- ensure that the code of your implementations is added to the classpath of the JPPF nodes, drivers and clients.

Note 1: Each concrete subclass of [JPPFCompositeSerialization](#) must have an implicit or explicit no-args **constructor**

Note 2: All JPPF built-in serializations are defined with the very same mechanism

To use one or more composite serializations, you need to specify them as a space-separated list in the value of the "jppf.object.serialization.class" property, in this format:

```
jppf.object.serialization.class = name1 ... nameN serialization_class_name
```

where:

- each *nameX* is the name assigned to a composite serialization
- *serialization_class_name* is the fully qualified class name of a serialization scheme as seen in the [previous section](#).

For instance, to use ZLIB compression on top of the predefined [DefaultJPPFSerialization](#), we would configure it as:

```
# ZLIB compression on top of JPPF serialization
jppf.object.serialization.class = ZLIB org.jppf.serialization.DefaultJPPFSerialization
```

Built-in composite serializations:

- [ZLIBSerialization](#) (name "ZLIB"): provides zlib compression and decompression based on the [java.util.zip.*](#) APIs
- [LZ4Serialization](#) (name "LZ4"): also provides compression and decompression, using the [LZ4 library](#). It is much faster than ZLIB but has a smaller compression rate.

Related sample:

The [DataEncryption](#) sample uses a composite serialization implementation to encrypt, via symmetric encryption, the objects transported over the network by JPPF.

7.1.3 Network interceptors

This extension allows user-defined code to intercept and make use of any JPPF-initiated network connection as soon as it is established. A typical use of this extension is so you can perform a custom authentication procedure before a connection is considered valid, but it is not limited to that.

The network interceptors are used with all possible types of network connections in JPPF: client to driver, node to driver, driver to driver, JMX connections and recovery connections (heart-beat mechanism) between nodes and drivers.

7.1.3.1 Creating a network interceptor

To define a network interceptor, simply create a class with a no-arg constructor that implements the interface [NetworkCommunicationInterceptor](#), which is defined like this:

```
public interface NetworkConnectionInterceptor {
    // The list of interceptors loaded via the Service Provider Interface
    List<NetworkConnectionInterceptor> INTERCEPTORS;
    // Called when a Socket is accepted by a ServerSocket
    // on the server side of a connection
    boolean onAccept(Socket acceptedSocket);
    // Called when a SocketChannel is accepted by a ServerSocketChannel
    // on the server side of a connection
    boolean onAccept(SocketChannel acceptedChannel);
    // Called when a Socket is connected on the client side of a connection
    boolean onConnect(Socket connectedSocket);
    // Called when a SocketChannel is connected on the client side of a connection
    boolean onConnect(SocketChannel connectedChannel);
}
```

Note that there are 2 `onAccept()` and 2 `onConnect()` methods: one takes a [Socket](#) as input, the other takes a [SocketChannel](#). `SocketChannels` are used by the JPPF driver for connections with nodes, clients or other servers. The JPPF driver makes sure they are configured in **blocking mode** before passing them to an interceptor.

Semantically speaking, `onAccept()` is called from the server side of a connection: the provided `Socket` or `SocketChannel` is obtained from a call to `ServerSocket.accept()` and `ServerSocketChannel.accept()`, respectively. On the other hand, `onConnect()` is called from the client side of a connection, that is, the side that initiates the connection.

In practice, [Socket](#) and [SocketChannel](#) use very different APIs for I/O reads and writes. However, when a `SocketChannel` is configured in blocking mode, it is possible to work directly with the underlying `Socket` via a prior call to `SocketChannel.socket()`. Consequently, you could write your interceptor like this:

```
public class MyInterceptor implements NetworkConnectionInterceptor {
    @Override
    public boolean onAccept(Socket acceptedSocket) {
        // do the job ...
        return ...;
    }

    @Override
    public boolean onConnect(Socket connectedSocket) {
        // do the job ...
        return ...;
    }

    @Override
    public boolean onAccept(SocketChannel acceptedChannel) {
        return onAccept(acceptedChannel.socket());
    }

    @Override
    public boolean onConnect(SocketChannel connectedChannel) {
        return onConnect(connectedChannel.socket());
    }
}
```

To make life even easier, JPPF provides the class [AbstractNetworkConnectionInterceptor](#), which implements the methods from [NetworkCommunicationInterceptor](#) as in the example above, and delegates the stream-based processing to two abstract methods. Our example could then be rewritten as:

```
public class MyInterceptor extends AbstractNetworkConnectionInterceptor {
    @Override
    public boolean onAccept(Socket acceptedSocket) {
        // do the job ...
        return ...;
    }

    @Override
    public boolean onConnect(Socket connectedSocket) {
        // do the job ...
        return ...;
    }
}
```

7.1.3.2 Plugging-in the interceptor

To discover the interceptor, JPPF uses the Service Provider Interface (SPI):

- in your sources or resources folder, create a file **org.jppf.comm.interceptor.NetworkConnectionInterceptor** in the **META-INF/services** folder
- in this file, add the fully qualified name of your interceptor implementation class, e.g.: org.example.MyInterceptor
- make sure the META-INF/services/org.jppf.comm.interceptor.NetworkConnectionInterceptor file is then placed into the resulting jar file
- add the jar file to the classpath of **all** JPPF components in your grid topology

7.1.3.3 Related sample

the [Network Interceptor demo](#) illustrates a simple encrypted authentication scheme implemented with an interceptor.

7.1.4 Environment providers for JMX remote connections

JPPF provides a management and monitoring API that relies on the [JMX remote](#) APIs and the JMXMP remote connector. This extension allows user-defined code to override or add parameters to those provided by default to both sides of each JMX connection. A possible use for this is to provide a way to use external tools such as JConsole or VisualVM to monitor JPPF servers and nodes. It can also be used to configure more sophisticated authentication and authorization protocols such as provided in the SASL profile of the JMXMP protocol. Of course, there can be many more uses.

To establish a JMX connection, JPPF uses the following standard JMX remote APIs:

- on the server side of the connection:
`JMXConnectorServerFactory.newJMXConnectorServer(JMXServiceURL serviceURL, Map<String,?> environment, MBeanServer mbeanServer)`
- on the client side of the connection:
`JMXConnectorFactory.newJMXConnector(JMXServiceURL serviceURL, Map<String,?> environment)`

You will note that both sides of the connection accept an `environment` parameter, which contains the parameters provided to the JMX remote connector. An environment provider allows you to define a different set of parameters that may override or add to these environment properties.

To achieve this, JPPF has two interfaces, one for each side of a JMX connection:

[ServerEnvironmentProvider](#) is defined like this:

```
public interface ServerEnvironmentProvider {  
    // Get a set of environment properties add to or override those  
    // passed to each new remote connector server  
    Map<String, ?> getEnvironment();  
}
```

[ClientEnvironmentProvider](#) is defined like this:

```
public interface ClientEnvironmentProvider {  
    // Get a set of environment properties add to or override those  
    // passed to each new remote connector  
    Map<String, ?> getEnvironment();  
}
```

To create an environment provider, you need to create a class with a no-args constructor, which implements one of these interfaces. For example, here is a simple client environment provider:

```
public class MyClientProvider implements ClientEnvironmentProvider {  
    @Override  
    public Map<String, ?> getEnvironment() {  
        Object myObject = ...;  
        Map<String, Object> env = new HashMap<>();  
        env.put("my.property", myObject);  
        return env;  
    }  
}
```

Note: for this code to compile, you need the `jppf-jmxremote_optional-x.y.z.jar` file in your build path.

As for many other extensions, environment providers are discovered via the Service Provider Interface (SPI) mechanism:

For a **server** environment provider:

- create a file `javax.management.remote.generic.ServerEnvironmentProvider` in the **META-INF/services** folder
- in this file, add the fully qualified class name of your implementation of [ServerEnvironmentProvider](#)

For a **client** environment provider:

- create a file `javax.management.remote.generic.ClientEnvironmentProvider` in the **META-INF/services** folder
- in this file, add the fully qualified class name of your implementation of [ClientEnvironmentProvider](#)

7.1.5 Monitoring data providers

A monitoring data provider is a service which provides diagnostics, health or information data about the (JVM) process of a JPPF node or driver, or about the machine it runs on. This data is provided to the [health monitoring MBean](#), and transported in each [health snapshot](#) that it supplies on demand.

The providers are also used by the JPPF desktop and web admin consoles, where they are rendered as columns in the JVM health view. The desktop console also makes the numeric values available as selectable fields for the charts.

7.1.5.1 Defining a provider

A monitoring data provider is defined as a subclass of the abstract class [MonitoringDataProvider](#), which specifies three abstract methods to override:

```
public abstract class MonitoringDataProvider {
    // Perform the initialization of this provider
    public abstract void init();

    // Perform the definition of the properties supplied by this provider
    public abstract void defineProperties();

    // Get the values for the defined properties
    public abstract TypedProperties getValues();

    ...
}
```

The `init()` method is invoked exactly once for each provider in a JPPF node or driver (producer side). It is intended for one-time initializations such as loading native libraries or reading resources files, etc. If no such initialization is needed in a provider, then the method's implementation should be left empty.

The method `defineProperties()` is called exactly once on both producer side, where the data comes from (drivers and nodes), and consumer side, where the data is rendered (web and desktop administration consoles). Each defined property has a name, a type, and optionally a validity range for the numeric properties. Individual properties are defined using the `setXXXProperty()` methods of [MonitoringDataProvider](#), where `xxx` represents the type of the property's value.

The `getValues()` method is intended to be called repeatedly on the producer side only. It performs the association between the names of the properties defined in `defineProperties()` and their actual value. This method returns the association as a [TypedProperties](#) object.

Important: a concrete implementation of [MonitoringDataProvider](#) must provide a no-args constructor, either implicit or explicit

As an example, let's define a [MonitoringDataProvider](#) that supplies the system time as a formatted string. We could implement it as follows:

```
public class SystemTimeProvider extends MonitoringDataProvider {
    // example: 2018-05-21 17:40:37
    private final SimpleDateFormat dtFormat = new SimpleDateFormat("YYYY-MM-dd HH:mm:ss");

    @Override
    public void init() { } // nothing to initialize

    @Override
    public void defineProperties() {
        // define the "systemTime" property as a string with an empty default value
        setStringProperty("systemTime", "");
    }

    @Override
    public TypedProperties getValues() {
        // set the current value of the "systemTime" property
        return new TypedProperties().setString("systemTime", dtFormat.format(new Date()));
    }
}
```

7.1.5.2 Value converters

A monitoring data provider can associate a converter for each property it defines, in order to produce a displayable string based on the raw value of the property. This displayable string will then be used by the administration consoles UIs, instead of those generated by default converters.

A value converter is an implementation of the [MonitoringValueConverter](#) interface, declared as follows:

```
public interface MonitoringValueConverter
// Convert or format the specified value expressed as a string
    String convert(String value);
}
```

The association of a value converter with a monitored property is performed with the method `setConverter(String, MonitoredValueConverter)` of [MonitoringDataProvider](#). The following example redefines our `SystemTimeProvider` implementation, such that the value of the property becomes the result of `System.currentTimeMillis()` (a long value), with an associated converter that transforms it into a readable date:

```
public class SystemTimeProvider extends MonitoringDataProvider {
    ...

    @Override
    public void defineProperties() {
        setLongProperty("systemTime", System.currentTimeMillis());
        // convert to a formatted date
        setConverter("systemTime", value -> dtFormat.format(new Date(Long.valueOf(value))));
    }

    @Override
    public TypedProperties getValues() {
        return new TypedProperties().setLong("systemTime", System.currentTimeMillis());
    }
}
```

[MonitoringValueConverter](#) also defines a number of specialized sub-interfaces and an implementation to conveniently convert typed data (int, long, float, double): [IntConverter](#), [LongConverter](#), [FloatConverter](#), [DoubleConverter](#) and [DoubleConverterWithFractionDigits](#). Accordingly, the `defineProperties()` method of our `SystemTimeProvider` can be further simplified:

```
public class SystemTimeProvider extends MonitoringDataProvider {
    ...

    @Override
    public void defineProperties() {
        setLongProperty("systemTime", System.currentTimeMillis());
        // convert to a formatted date
        setConverter("systemTime", (LongConverter) value -> dtFormat.format(new Date(value)));
    }
}
```

7.1.5.3 Plugging a monitoring data provider

Monitoring data providers are discovered via the [Service Provider Interface \(SPI\)](#) mechanism. To plug a provider into JPPF, proceed as follows:

- In your source directory, or in a separate resources directory, create the service file:

```
META-INF/services/org.jppf.management.diagnostics.provider.MonitoringDataProvider
```

- Open this file in a text editor and add a line with the fully qualified name of your provider implementation. For instance, for the `SystemTimeProvider` example in the previous section, we would add a line with the following content:

```
test.SystemTimeProvider
```

Note: you can add as many provider implementations as you wish.

- Create a jar file which contains this service file, along with your provider implementation and its support classes

- To make the provider available to the drivers and nodes, just add the jar file to each driver's classpath. The nodes will automatically lookup and download the provider implementations from the driver, using their distributed classloader.

- To make the provider available to a desktop administration console, just add the jar file to the console's classpath. The provider's properties will be shown as additional columns in the "JVM health" view of the console. The numeric properties will also be added to the list of available fields in the charts configuration dialog.

- To make the provider available to a web administration console, you will need to repackage the war file so as to add the provider jar to its WEB-INF/lib folder

7.1.5.4 Localization support

If you have integrated a monitoring data provider in one of the administration consoles, you may have noticed that the corresponding column name in the "JVM health" view is exactly the name of the property, which may be more or less readable. The column header tooltip will also display the property name. If you need something more readable, based on the user's locale, follow these steps:

- in your provider implementation, override the `getLocalizationBase()` method of the [MonitoringDataProvider](#) class, to specify the base name of the localization files. For the `SystemTimeProvider` example, we could do as follows:

```
@Override
protected String getLocalizationBase() {
    return "test.SystemTimeProvider";
}
```

The returned string is used as the prefix for one or more properties files, based on the locale. For example the default translation would be found in the `test.SystemTimeProvider.properties` file, whereas the translation for the `en_GB` locale would be found in `test.SystemTimeProvider_en_GB.properties`.

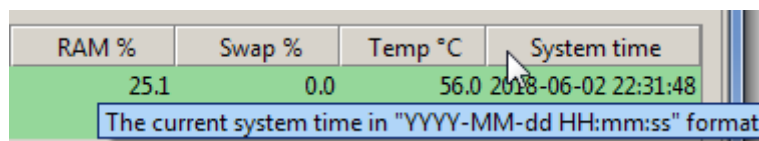
- write a default translation file, where you can specify the column name and associated tooltip as follows:

```
property_name = some short translation
property_name.doc = a longer localized tooltip text \n possibly on multiple lines
```

The file name must be the string returned by `getLocalizationBase()`, with the `.properties` extension, for example: `test.SystemTimeProvider.properties`, with the following content:

```
# column name in JVM health view
systemTime = System time
# column header tooltip (.doc suffix)
systemTime.doc = The current system time in "YYYY-MM-dd HH:mm:ss" format
```

The screenshot below shows how this is rendered in the desktop console:



RAM %	Swap %	Temp °C	System time
25.1	0.0	56.0	2018-06-02 22:31:48

- finally, do the same for all locales or languages for which you need a translation.

7.1.5.5 Accessing the supplied values

The values supplied by the [MonitoringDataProvider](#) can be accessed via [HealthSnapshot](#) objects, which are obtained from the [DiagnosticsMBean](#) management bean. For more information on the `DiagnosticsMBean` and how to use it, please read the documentation on [JVM health monitoring](#).

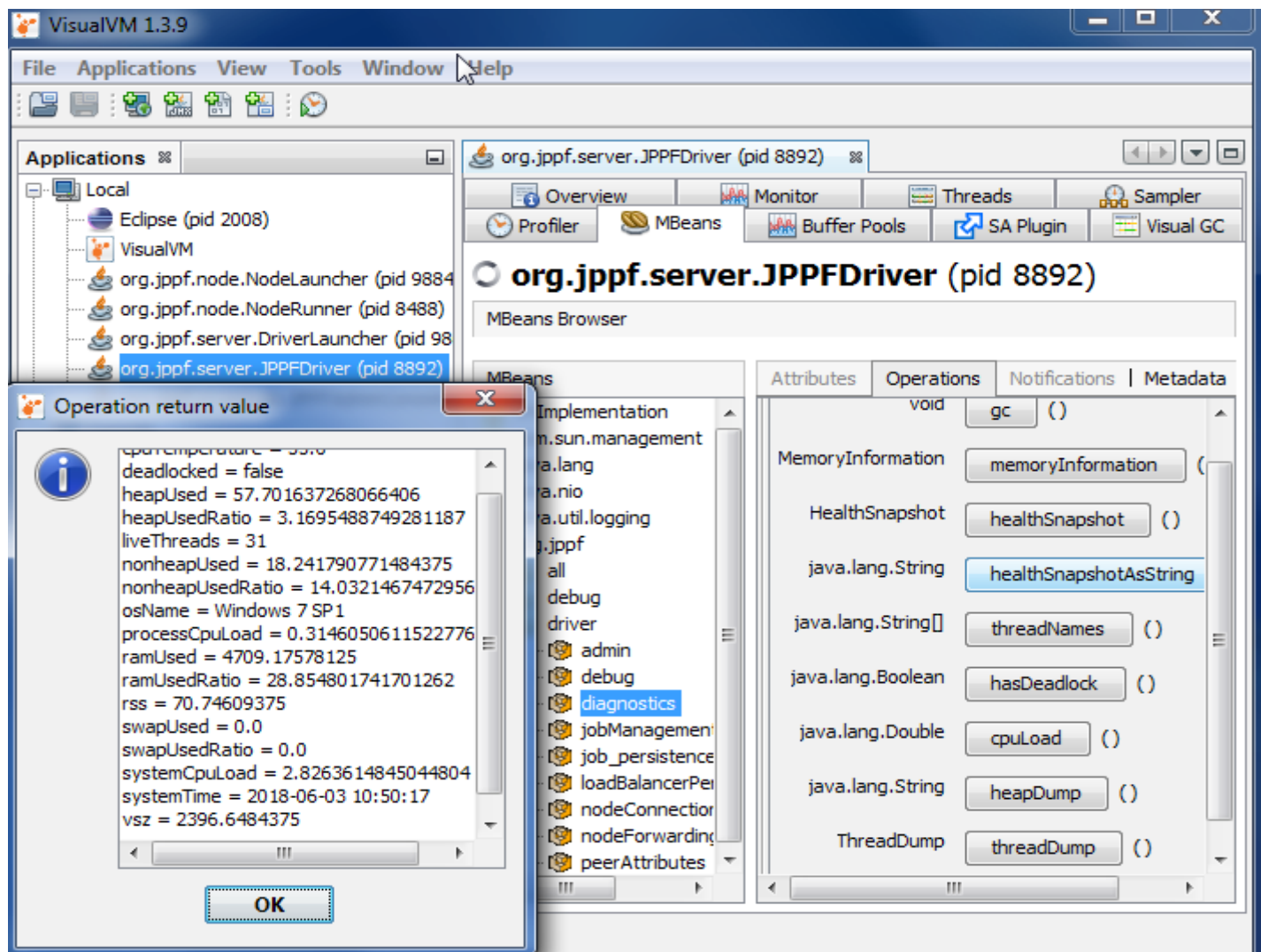
From a [HealthSnapshot](#), individual values can be obtained with one of the `getXXX(String)` methods, where `XXX` represents the desired type of the value. For example:

```
try (JMXDriverConnectionWrapper driverJmx =
    new JMXDriverConnectionWrapper("my.driver.host", 11111)) {
    // wait up to 5s for the JMX connection to the driver
    driverJmx.connectAndWait(5000L);
    // obtain a proxy to the DiagnosticsMBean
    DiagnosticsMBean diag = driverJmx.getDiagnosticsProxy();
    // request a snapshot
    HealthSnapshot snapshot = diag.healthSnapshot();
    // print the value of "systemTime"
    System.out.println("the driver system time is " + snapshot.getString("systemTime"));
} catch (Exception e) { e.printStackTrace(); }
```

You may also obtain all the values as a [TypedProperties](#) object with the snapshot's `getProperties()` method:

```
DiagnosticMBean diag = ...;
HealthSnapshot snapshot = diag.healthSnapshot();
// get all properties and print the value of "systemTime"
TypedProperties properties = snapshot.getProperties();
System.out.println("the driver system time is " + properties.getString("systemTime"));
```

Lastly, you can also use the `DiagnosticMBean`'s [healthSnapshotAsString\(\)](#) method, which provides a string with all properties and their values, in a format compatible with properties files. This allows you to view the snapshot from external tools such as JConsole or VisualVM, as illustrated in this screenshot:



7.2 Drivers and nodes

7.2.1 Pluggable MBeans

Developers can write their own management beans (MBeans) and register them with the 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.

7.2.1.1 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 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

7.2.1.2 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.

7.2.1.2.1 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`:

```
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();
    }
}
```

7.2.1.2.2 Implement the node MBean provider interface

To make our MBean pluggable to the nodes, it must be recognized as a corresponding service instance. To this effect, we will create an implementation of the interface [JPPFNodeMBeanProvider](#), which will provide the node with enough information to create the MBean and register it with the MBean server. This interface is defined as follows:

```
// service provider interface for pluggable management beans for JPPF nodes
public interface JPPFNodeMBeanProvider extends JPPFMBeanProvider {
    // return a concrete MBean instance whose class must implement the interface defined by
    // JPPFMBeanProvider.getMBeanInterfaceName()
    public Object createMBean(Node node);
}
```

Please note that the parameter passed to `createMBean(Node)` allows the MBean to access information on the node via the [Node](#) API.

As we can see, this interface declares a single method whose role is to create an instance of our MBean implementation. There is no obligation to use the node parameter, it is provided here because the JPPF built-in node MBean use it. As stated in the method comment, the class of the created object must implement an MBean interface, whose name is given by the method `getMBeanInterfaceName()` in the super-interface [JPPFMBeanProvider](#), defined as follows:

```
// service provider interface for pluggable management beans
public interface JPPFMBeanProvider {
    // get the fully qualified name of the management interface defined by this provider
    public String getMBeanInterfaceName();

    // get the name of the specified MBean, 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:

```
// 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(Node 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";
    }
}
```

7.2.1.2.3 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
```

7.2.1.2.4 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.

7.2.1.2.5 Using the MBean

We can now write a simple class to test our new custom MBean:

```

package org.jppf.example.node.test;

import org.jppf.management.JMXNodeConnectionWrapper;

// simple class to test a custom 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");
    }
}

```

7.2.1.3 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.

7.2.1.3.1 Create the MBean interface and its implementation

We will simply reuse the interface `AvailableProcessorsMBean` and its implementation `AvailableProcessors` that we have already created.

7.2.1.3.2 Implement the server MBean provider interface

This time , we will implement the interface [JPPFDriverMBeanProvider](#), defined as follows:

```

public interface JPPFDriverMBeanProvider extends JPPFMBeanProvider {
    // Return a concrete MBean instance
    Object createMBean();

    // Return a concrete MBean
    default Object createMBean(JPPFDriver driver) {
        return createMBean();
    }
}

```

The JPPF driver will always call `createMBean(JPPFDriver)` when creating a custom MBean. Note that its default implementation merely delegates to `createMBean()` without using the parameter. Implementations that wish to use the [JPPFDriver](#) API should override this method.

Our server MBean provider implementation is like this:

```

// AvailableProcessors MBean provider
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`.

7.2.1.3.3 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
```

7.2.1.3.4 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.

7.2.1.3.5 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");
    }
}
```


7.2.2 Startup classes

Startup classes allow a piece of code to be executed at startup time of a node or server. They can be used for many purposes, including initialization of resources such as database connections, JMS queues, cache frameworks, authentication, etc ... They permit the creation of any object within the same JVM as the JPPF component they run in.

Startup classes are defined using the Service Provider Interface. The general workflow to create a custom startup class is as follows:

- step 1: create a class implementing the startup class provider interface
- step 2: add or update the corresponding service definition file in the META-INF/services folder
- step 3: create a jar file containing the above elements and deploy it in the node or server class path

This mechanism relies on the following rules:

- the provider interface for a node or server startup class extends the interface [JPPFStartup](#), which itself extends [java.lang Runnable](#). Thus, writing a startup class consists essentially in writing code in the `run()` method.
- the provider interface implementation must have a no-arg constructor
- startup classes are instantiated and run just after the JPPF and custom MBeans have been initialized. This allows a startup class to subscribe to any notifications that an MBean may emit.

Related sample: “Startup Classes” sample in the JPPF samples pack.

7.2.2.1 Node startup classes

7.2.2.1.1 Implement the node startup class provider interface

To make our startup class pluggable to the nodes, it must be recognized as a corresponding service instance. To this effect, we will create an implementation of the interface [JPPFNodeStartupSPI](#), which will provide the node with enough information to create and run the startup class. This interface is defined as follows:

```
public interface JPPFNodeStartupSPI extends JPPFStartup { }
```

As we can see, this is just a marker interface, used to distinguish between node startup classes and server startup classes. As an example, we will create an implementation that simply prints a message when the node starts:

```
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 {
    @Override public void run() {
        System.out.println("I'm a node startup class");
    }
}
```

A server startup implementation may need information on the JPPF node itself. To this effect, you can define a method with this exact signature: **public void setNode(Node)**. This method, if defined, will be called exactly once, before the `run()` method. It provides access to information on the node via the [Node](#) API. It can be used as follows:

```
public class TestNodeStartup implements JPPFNodeStartupSPI {
    private Node node;

    @Override public void run() {
        System.out.println("Node configuration: " + node.getConfiguration());
    }

    // the JPPF node will detect that this method exists and call it
    public void setNode(Node node) {
        this.node = node;
    }
}
```

7.2.2.1.2 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
```

7.2.2.1.3 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.*

7.2.2.2 Server startup classes

7.2.2.2.1 Implement the server startup class provider interface

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:

```
import org.jppf.startup.JPPFNodeStartupSPI;

// This is a test of a server startup class
public class TestDriverStartup implements JPPFDriverStartupSPI {
    @Override
    public void run() {
        System.out.println("I'm a server startup class");
    }
}
```

A server startup implementation may need information on the JPPF driver itself. To this effect, you can define a method with this exact signature: **public void setDriver(JPPFDriver)**. This method, if defined, will be called exactly once, before the `run()` method. It allows implementers to access information on the driver via the [JPPFDriver](#) API. It can be used as follows:

```
public class TestDriverStartup implements JPPFDriverStartupSPI {
    private JPPFDriver driver;

    @Override public void run() {
        System.out.println("Server configuration: " + driver.getConfiguration());
    }

    // the JPPF driver will detect that this method exists and call it
    public void setDriver(JPPFDriver driver) {
        this.driver = driver;
    }
}
```

7.2.2.2.2 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
```

7.2.2.2.3 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.

7.2.3 Pluggable MBeanServerForwarder

JPPF provides the ability to set an [MBeanServerForwarder](#) on all JMX remote servers created by the drivers and nodes. This can be used, for example, to implement an authorization mechanism that will grant or deny access to operations on the MBeans.

A forwarder implements the [MBeanServerForwarder](#) interface and is declared in a driver's or node's configuration file as follows:

```
jppf.management.server.forwarder = my.forwarder.Implementation param1 ... paramN
```

The value of the property is a set of space-separated strings, where the first string is the fully qualified class name of an implementation of `MBeanServerForwarder`, and the remaining strings are optional parameters passed on to the forwarder at construction time.

For the parameters to be passed on, the forwarder must also implement either of:

- a constructor that takes a `String[]` argument
- a `setParameters(String[])` method

If both are implemented, then only the constructor will be used. If none is implemented, then no parameter will be passed, even if they are declared in the configuration. Additionally, a no-arg constructor is then expected.

In practice, it may be more convenient to extend the class [MBeanServerForwarderAdapter](#), which implements all the methods of the [MBeanServerForwarder](#) interface and merely delegates to the underlying `MBeanServer` if it has been set. It also provides a getter and setter for the parameters, so you don't have to implement a `setParameters(String[])` method either:

```
public class MBeanServerForwarderAdapter implements MBeanServerForwarder {  
    // Set the parameters defined in the configuration, if any  
    public void setParameters(final String[] parameters)  
  
    // Get the parameters defined in the configuration, if any  
    public String[] getParameters()  
  
    // ... methods from the MBeanServerForwarder interface ...  
}
```

Note: each JMX server created in a driver or node has a distinct `MBeanServerForwarder` instance.

To get access to a configured [MBeanServerForwarder](#), you first need to access the [JMXServer](#) that encloses it:

- in a driver, use [JPPFDriver.getInstance\(\).getJMXServer\(boolean secure\)](#)
- in a node, obtain a reference to the [Node](#), then use `Node.getJmxServer()`

7.3 Drivers and clients

7.3.1 Creating a custom load-balancer

Related sample: “[CustomLoadBalancer](#)” in the JPPF samples pack.

7.3.1.1 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 dispatched 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);

    // 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 1: all JPPF built-in load balancing algorithms are implemented and plugged-in as custom algorithms

Note 2: for a fully detailed explanation of how load balancers work in JPPF, please read the [Load Balancing](#) section.

7.3.1.2 Implementing the algorithm and its profile

First let's implement our parameters profile. To this effect, we implement the interface [LoadBalancingProfile](#), which is merely a marker interface (it does not define any method):

```
public interface LoadBalancingProfile extends Serializable {  
}
```

As we can see, this interface has a single method that creates a copy of a profile. Now let's see how it is implemented in the [FixedSizeProfile](#) class:

```
// Profile for the fixed bundle size load-balancing algorithm  
public class FixedSizeProfile implements LoadBalancingProfile {  
    // The bundle size  
    private int size = 1;  
  
    // Default constructor  
    public FixedSizeProfile() {  
    }  
  
    // Initialize this profile with values read from the specified configuration  
    public FixedSizeProfile(TypedProperties config) {  
        size = config.getInt("size", 1);  
    }  
  
    // Get the bundle size  
    public int getSize() {  
        return size;  
    }  
  
    // Set the bundle size  
    public void setSize(int size) {  
        this.size = size;  
    }  
}
```

This implementation is fairly trivial, the only notable element being the constructor taking a [TypedProperties](#) parameter, which will allow us to read the size parameter from the JPPF configuration file.

Now let's take a look at the algorithm implementation itself:

```
public class FixedSizeBundler extends AbstractBundler {  
    // Initialize this bundler  
    public FixedSizeBundler(LoadBalancingProfile profile) {  
        super(profile);  
    }  
  
    // This method always returns a statically assigned bundle size  
    public int getBundleSize() {  
        return ((FixedSizeProfile) profile).getSize();  
    }  
  
    // Get the max bundle size that can be used for this bundler  
    protected int maxSize() {  
        return -1;  
    }  
}
```

The first thing we can notice is that the `feedback()` method is not even implemented! This is due to the fact that our algorithm is independent from the context and involves no computation. Thus, we use the default implementation in `AbstractBundler`, which does nothing. This is visible in the `getBundleSize()` method, where we simply return the value provided in the parameters profile.

We also notice a new method named `maxSize()`. It returns a value representing the maximum bundle size that a bundler can use at a given time. The goal of this is to avoid that a node receives all or most of the tasks, while the other nodes would not receive anything and thus would have nothing to do. This method is declared in the abstract class `AbstractBundler` and doesn't have any default implementation, to avoid any tight coupling between the bundler and the

environment in which it runs. This allows the bundler to be used outside of the JPPF server, as is done for instance in the JPPF client when local execution mode is used along with remote execution.

In the context of the server, we have found that an efficient value for `maxSize()` can be computed from the current maximum number of tasks among all the jobs in the server queue. This value is accessible by calling the method [JPPFQueue.getMaxBundleSize\(\)](#). We could then rewrite our `maxSize()` method as follows:

```
protected int maxSize() {
    return JPPFDriver.getQueue().getMaxBundleSize() / 2;
}
```

The algorithm could then determine that a node should not receive more than half of that value (or 75% or any other function of it, whatever is deemed more efficient), so that other nodes will not be idle and the overall throughput will be optimized.

Tip: if your algorithm depends on the number of nodes, you can use a bundler instances count as a static variable in your implementation, and use the `setup()` and `dispose()` methods to increment and decrement the count as needed. For instance:

```
private static AtomicInteger instanceCount = new AtomicInteger(0);

public void setup() {
    instanceCount.incrementAndGet();
}

public void dispose() {
    instanceCount.decrementAndGet();
}
```

7.3.1.3 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);
}
}
```

7.3.1.4 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.load.balancer.spi.JPPFBundlerProvider, 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.load.balancer.spi.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.

7.3.1.5 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.

7.3.1.6 Job-aware load balancers

Load-balancers can gain access to information on a job via the [JPPFDistributedJob](#) interface. 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 information
    JPPFDistributedJob getJob();
    // Set the current job
    void setJJob(JPPFDistributedJob job);
}
```

The method setJob() 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.

An example usage of [JobAwareness](#) can be found in the [CustomLoadBalancer](#) sample, in the JPPF samples pack.

7.3.2 Custom load-balancer state persistence

Many of the existing load-balancing algorithms are adaptive and go through a convergence phase before reaching a state of optimal efficiency. Unfortunately, this state disappears whenever a node, driver or client involved in load-balancing is stopped.

The load-balancer persistence allows storing the state of load-balancer instances and restoring them whenever a node reconnects to a driver (server-side load-balancing) or a client reconnects to a driver (client-side load-balancing).

7.3.2.1 Identification of persisted load-balancer states

Persistence and restoration of a load-balancer state requires an identifier that is both unique and repeatable across reconnection and restart of the peers involved (node + driver or driver + client). Using combinations of the components' UUIDs will not work here, since UUIDs only exist as long as each component's JVM is alive and are generated anew at each restart.

Instead, JPPF computes a resilient identifier which includes the IP address of both parties, the related driver's port and other properties found in the JPPF component's configuration. A load-balancer state is further identified by the name of the of its algorithm. For normalization purposes, both parts of the identifier are transformed using a hash function. This function defaults to SHA-1 and can be changed using the following configuration property:

```
jppf.load.balancing.persistence.hash = SHA-1
```

7.3.2.2 PersistentState interface

Any load-balancer implementation that wishes to have its state persisted must implement the [PersistentState](#) interface, defined as follows:

```
public interface PersistentState {  
    // Get this load-balancer's state  
    Object getState();  
  
    // Set this load-balancer's state  
    void setState(Object state);  
  
    // Set this load-balancer's state  
    Lock getStateLock();  
}
```

Any [Bundler](#) implementation which also implements this interface will see its state, as provided by the `getState()` method, persisted when load-balancer persistence is enabled. The `setState()` method is be called by a JPPF driver or client upon the load-balancer's initialization, when the load-balancer's state is restored from the persistence store.

State synchronization:

Care must be taken to synchronize access to the bundler's state. The persistence facility will first serialize it, before storing it. Keep in mind that the serialization process will traverse any data structure owned by the bundler's state. At the same time, especially when the persistence is asynchronous, the bundler may still receive updates from another thread via its `feedback()` method and modify its state.

This means that bundler persistence introduces a potential race condition. When the bundler state uses collections, this will typically lead to `ConcurrentModificationException` errors and prevent the persistence from working, or even corrupt it.

To avoid this, both the load-balancer and the persistence implementation must synchronize on the state. This the very reason why [PersistentState](#) has a `getStateLock()` method, to provide a lock that both load-balancer and persistence can use for safe access tot he state.

For instance, all the built-in JPPF load-balancers that support persistence implement a pattern similar to the one below:

```
public class MyBundler extends AbstractAdaptiveBundler<MyProfile>  
    implements PersistentState {  
    // The state of this bundler  
    private final MyBundlerState state;  
    // The load-balancer state lock  
    private final Lock lock = new ReentrantLock();
```



```

public MyBundler(MyProfile profile) {
    super(profile);
    state = new MyBundlerState();
}

@Override public void feedback(int size, double time) {
    lock.lock();
    try {
        // ... compute the new bundle size ...
    } finally {
        lock.unlock()
    }
}

@Override public Object getState() {
    lock.lock();
    try {
        return state;
    } finally {
        lock.unlock()
    }
}

@Override public void setState(Object o) {
    BundlerState other = (BundlerState) o;
    lock.lock();
    try {
        state.bundleSize = other.bundleSize;
        state.performanceCache = other.performanceCache;
    } finally {
        lock.unlock()
    }
}

@Override public Lock getStateLock() {
    return lock;
}

// Holds the state of this bundler for persistence
private static class BundlerState implements Serializable {
    private int bundleSize = 1;
    private PerformanceCache performanceCache = new PerformanceCache();
}
}

```

7.3.2.3 LoadBalancerPersistence interface

A custom load-balancer persistence must implement the [LoadBalancerPersistence](#) interface, defined as follows:

```

public interface LoadBalancerPersistence {
    // Load the state of a load balancer from the persistence store
    Object load(LoadBalancerPersistenceInfo info) throws LoadBalancerPersistenceException;

    // Store a load balancer state to the persistence store
    void store(LoadBalancerPersistenceInfo info) throws LoadBalancerPersistenceException;

    // Delete the specified load-balancer state(s) from the persistence store
    void delete(LoadBalancerPersistenceInfo info) throws LoadBalancerPersistenceException;

    // Retrieve the specified channel or algorithm IDs from the persistence store
    List<String> list(LoadBalancerPersistenceInfo info)
        throws LoadBalancerPersistenceException;
}

```

All the methods of this interface are called by the JPPF load-balancer persistence facility, during the load-balancers life cycle, but also during user-initiated management requests. As we also can see, all the methods take a single parameter of type [LoadBalancerPersistenceInfo](#), which identifies the artifact(s) to store, retrieve or delete. It is defined as follows:

```

public class LoadBalancerPersistenceInfo {
    // Get a readable identifier for the channel, resilient over restarts of the processes
    // involved in the load-balancing
    public String getChannelString()
    // Get the channel identifier, a hash of the string returned by getChannelString()
    public String getChannelID()
    // Get the readable name of the related load-balancing algorithm
    public String getAlgorithm()
    // Get a hash of the load-balancing algorithm name obtained via getAlgorithm()
    public String getAlgorithmID()
    // Get the load-balancer state
    public Object getState()
    // Get a lock used to synchronize access to the state
    public Lock getStateLock()
    // Serialize the state into an array of bytes
    public byte[] getStateAsBytes() throws Exception
    // Serialize the state into an output stream
    public void serializeToStream(final OutputStream stream) throws Exception
}

```

The methods `getChannelString()` and `getAlgorithm()` provide human-readable versions of the identifiers for the channel and algorithm. They are here for logging and debugging purposes only and it is not recommended to use them to actually identify load-balancer states in the persistence store.

The `getStateLock()` method returns the same `Lock` object as the one provided by the `getStateLock()` method of [PersistentState](#). In principle, it should only be used in the `store()` and `load()` methods of [LoadBalancerPersistence](#), and it will return `null` when passed to the `list()` and `delete()` methods of `LoadBalancerPersistence`.

The `getStateAsBytes()` and `serializeToStream()` methods are convenience methods that serialize the load-balancer state to a byte array or a stream, respectively. They also abstract two characteristics of the serialization:

- internally, they synchronize on the `Lock` object provided by `getStateLock()`
- they also perform the serialization according to the serialization scheme configured in JPPF

Back to the [LoadBalancerPersistence](#) interface, the specification for the `list()` and `delete()` methods is tightly coupled to the content of their [LoadBalancerPersistenceInfo](#) parameter.

For **`list(LoadBalancerPersistenceInfo info)`**:

- if `info` is null or both `info.getChannelID()` and `info.getAlgorithmID()` are null, then all channel IDs in the persistence store are returned
- if only `info.getAlgorithmID()` is null, then all the algorithm IDs for the specified channel are returned
- if only `info.getChannelID()` is null, then the IDs of the channels that have a persisted state for the algorithm are returned
- if neither `info.getChannelID()` nor `info.getAlgorithmID()` are null, then the specified algorithm ID is returned (list with a single entry) if the channel has an entry for it, otherwise an empty list must be returned

For **`delete(LoadBalancerPersistenceInfo info)`**:

- if `info` is null or both `info.getChannelID()` and `info.getAlgorithmID()` are null, then all entries in the persistence store are deleted
- if only `info.getAlgorithmID()` is null, then the states of all algorithm for the specified channel are deleted
- if only `info.getChannelID()` is null, then the states of the specified algorithm are deleted for all the channels
- if neither `info.getChannelID()` nor `info.getAlgorithmID()` are null, then only the state of the specified algorithm for the channel is deleted

Typically, a load-balancer persistence implementation will have the following structure:

```

public class MyPersistence implements LoadBalancerPersistence {
    public MyPersistence(String...params) {
        // process the parameters
    }

    @Override public Object load(LoadBalancerPersistenceInfo info)
        throws LoadBalancerPersistenceException {
        return ...;
    }

    @Override public void store(LoadBalancerPersistenceInfo info)
        throws LoadBalancerPersistenceException {
        // store the load-balancer state
    }
}

```

```

}

@Override public void delete(LoadBalancerPersistenceInfo info)
    throws LoadBalancerPersistenceException {
    if ((info == null) ||
        ((info.getChannelID() == null) && (info.getAlgorithmID() == null))) {
        // delete all entries in the store
    } else if (info.getAlgorithmID() == null) {
        // delete all entries for the channel with info.getChannelID()
    } else if (info.getChannelID() == null) {
        // delete all info.getAlgorithmID() entries for the channels that have it
    } else {
        // delete the [info.getChannelID(), info.getAlgorithmID()] entry
    }
}

@Override public List<String> list(LoadBalancerPersistenceInfo info)
    throws LoadBalancerPersistenceException {
    List<String> results = new ArrayList<>();
    if ((info == null) ||
        ((info.getChannelID() == null) && (info.getAlgorithmID() == null))) {
        // retrieve all channelIDs in the store
    } else if (info.getAlgorithmID() == null) {
        // retrieve all algorithmIDs for the channel with info.getChannelID()
    } else if (info.getChannelID() == null) {
        // delete all channelIDs that have an entry for info.getAlgorithmID()
    } else {
        // return info.getAlgorithmID() if the channel with info.getChannelID() has it
    }
    return results;
}
}

```

7.3.2.4 Configuration

Load-balancer persistence is setup via the JPPF configuration, in the following format:

```
jppf.load.balancing.persistence = <persistence_class_name> [param1 ... paramN]
```

As we can see, optional parameters can be passed on to an implementation from the configuration. To receive them, the implementation class must declare either a constructor that takes a `vararg String...` parameter, or a public `void setParameters(String...params)` method. The space-separated parameters can be used to specify the root directory for a file-based implementation, or JDBC connection parameters, but are not limited to these.

An implementation that does not declare a constructor with a `String...` argument *must* declare a no-args constructor. On the other hand, if a persistence implementation declares both a constructor with a `String...` argument and a `setParameters()` method, then the constructor will always be preferred.

When the **"jppf.load.balancing.persistence"** property is unspecified, then load-balancer persistence is disabled.

7.3.2.5 Reference

We invite you to read the [reference section](#) on load-balancer state persistence, including the details on the JPPF [built-in implementations](#).

7.4 Drivers

7.4.1 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. These notifications can be received either locally from within the server JVM, or remotely via a JMX listener.

7.4.1.1 Local notifications

As for other JPPF extensions, this one 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.

7.4.1.2 JMX notifications

JMX notifications of node connection and disconnection events are sent by a server MBean which implements the interface [JPPFNodeConnectionNotifierMBean](#), defined as follows:

```
public interface JPPFNodeConnectionNotifierMBean extends NotificationEmitter {
    // The name of this MBean, used when it is registered with an MBean server
    String MBEAN_NAME = "org.jppf:name=nodeConnectionNotifier,type=driver";

    // The type of notification which indicates that a node is connected
    String CONNECTED = "connected";

    // The type of notification which indicates that a node is disconnected
    String DISCONNECTED = "disconnected";
}
```

As we can see, this MBean doesn't have any method or attribute: it only sends notifications. The notifications received from this MBean are of the standard type [Notification](#) and should be interpreted as follows:

- the notification's `getUserObject()` method returns a [JPPFManagementInfo](#) object and the return value can be safely cast to this type.
- the `getType()` method returns one of the two dedicated constants defined in [JPPFNodeConnectionNotifierMBean](#): `CONNECTED` or `DISCONNECTED`.
- the `getSource()` method returns the value of the `MBEAN_NAME` constant.

Here is an example of notification listeners that processes node connection events via JMX:

```
public class MyNodeConnectionListener implements NotificationListener {
    @Override
    public void handleNotification(Notification notif, Object handback) {
        // retrieve the information on the node
        JPPFManagementInfo info = (JPPFManagementInfo) notif.getUserObject();
        switch (notif.getType()) {
            case JPPFNodeConnectionNotifierMBean.CONNECTED:
                // process node connected event ...
                break;
            case JPPFNodeConnectionNotifierMBean.DISCONNECTED:
                // process node disconnected event ...
                break;
        }
    }
}
```

All that remains to do is to register this listener with a JMX connection, as in this example:

```
// create the client
try (JPPFClient client = new JPPFClient()) {
    // obtain a working JMX connection to the server
    JMXDriverConnectionWrapper jmx =
        client.awaitWorkingConnectionPool().awaitWorkingJMXConnection();
    // register the JMX notification listener
    NotificationListener myListener = new MyNodeConnectionListener();
    jmx.addNotificationListener(JPPFNodeConnectionNotifierMBean.MBEAN_NAME, myListener);
    ...
} catch (Exception e) {
    e.printStackTrace();
}
```

7.4.2 Receiving the status of tasks dispatched to or returned from the nodes

7.4.2.1 Implementation

Each time a set of tasks is dispatched to, or returns from a node, the driver emits a notification which encapsulates information about the set of tasks, along with detailed information on each of the tasks it contains. To receive these notifications, you must write a listener which implements the [JobTasksListener](#) interface, defined as:

```
public interface JobTasksListener {  
    // Called when tasks from a job are dispatched to a node  
    void tasksDispatched(JobTasksEvent event);  
  
    // Called when tasks from a job return from the node  
    void tasksReturned(JobTasksEvent event);  
  
    // Called when tasks are about to be sent back to the client  
    void resultsReceived(JobTasksEvent event);  
}
```

Please also note that the methods of `JobTasksListener` will be called even if the client that submitted the job is disconnected, for any reason. Combined with the ability to process the tasks results, this enables a callback implemented on the server side, which can be used with a "submit and forget" strategy on the client side.

As we can see, all the methods receive events of type [JobTasksEvent](#), defined as follows:

```
public class JobTasksEvent extends EventObject {  
    // Get the uuid of the job to which the tasks belong  
    public String getJobUuid()  
    // Get the name of the job to which the tasks belong  
    public String getJobName()  
    // Get the job's server-side SLA  
    public JobSLA getJobSLA()  
    // Get the job metadata  
    public JobMetadata getJobMetadata()  
    // Get the list of tasks that were dispatched or returned  
    public List<ServerTaskInformation> getTasks()  
    // Get the reason why the set of tasks was returned by a node  
    public JobReturnReason getReturnReason()  
    // Get the information on the node where the tasks were dispatched or returned  
    public JPPFManagementInfo getNodeInfo()  
}
```

The method `getReturnReason()` provides a high-level indication of why the tasks were returned, among the possible reasons defined in the [JobReturnReason](#) enum:

```
public enum JobReturnReason {  
    // The tasks were normally processed by the node  
    RESULTS_RECEIVED,  
    // The processing of the tasks took longer than the specified dispatch timeout  
    DISPATCH_TIMEOUT,  
    // An error occurred in the node which prevented the normal execution of the tasks  
    NODE_PROCESSING_ERROR,  
    // An error occurred in the driver while processing the results returned by the node  
    DRIVER_PROCESSING_ERROR,  
    // The connection between node and server was severed before results could be returned  
    NODE_CHANNEL_ERROR  
}
```

The job return reason is only available for `tasksReturned()` and `resultsReceived()` notifications. Therefore, for a `tasksDispatched()` notification, `getReturnReason()` always returns `null`. Also please note that for a `resultsReceived()` notification the return reason is always `RESULTS_RECEIVED`.

The method `getTasks()` returns a list of [ServerTaskInformation](#) objects, providing details on individual tasks:

```
public class ServerTaskInformation implements Serializable {
    // Get the position of this task within the job submitted by the client
    public int getJobPosition()
    // Get the throwable raised during the processing of the task
    public Throwable getThrowable()
    // Get the number of times a dispatch of the task has expired
    public int getExpirationCount()
    // Get the maximum number of times the task can be resubmitted
    public int getMaxResubmits()
    // Get the number of times the task was resubmitted
    public int getResubmitCount()
    // Get the task result as binary data
    public InputStream getResultAsStream() throws Exception
    // Get the task result as a JPPF Task object
    public Task<?> getResultAsTask() throws Exception
}
```

Important: when using the `getResultAsTask()` method, you **must** ensure that the code of the task and all its dependencies are in the driver's classpath. This is because this method actually deserializes the task from the binary format in which it is stored in the driver.

Deserializing a task from a stream: in scenarios where, instead of directly deserializing the task, you wish to copy its binary form for later processing by a separate process, you will need a way to deserialize it that takes into account the configured [serialization scheme](#) and eventual [composite serialization](#). This can be easily achieved with the [JPPFSerialization.Factory](#) API, as in this example:

```
InputStream taskInputStream = ...;
JPPFSerialization serialization = JPPFSerialization.Factory.getSerialization();
Task<?> task = (Task<?>) serialization.deserialize(taskInputStream);
```

The following code is an example listener which prints out its events to the console:

```
public class MyJobTasksListener implements JobTasksListener {
    public MyJobTasksListener() {
        System.out.println("in MyJobTasksListener()"); // displayed at driver startup time
    }

    @Override
    public void tasksDispatched(TaskReturnEvent event) {
        printEvent(event);
    }

    @Override
    public void tasksReturned(TaskReturnEvent event) {
        printEvent(event);
    }

    @Override
    public void resultsReceived(TaskReturnEvent event) {
        printEvent(event);
    }

    private void printEvent(JobTasksEvent event) {
        System.out.printf("listener event: name=%s, uuid=%s, reason=%s, node=%s, %d tasks%n",
            event.getJobName(), event.getJobUuid(), event.getReturnReason(),
            event.getNodeInfo(), event.getTasks().size());
    }
}
```

and here is an example output:

```
listener event: name=my job, uuid=job_uuid, reason=RESULTS_RECEIVED,
node=JPPFManagementInfo[192.168.1.24:12001, type=node|MASTER, local=false, secure=false,
uuid=node_uuid], 5 tasks
```


7.4.2.2 Deployment / Integration

There are two ways to integrate a [JobTasksListener](#) with the JPPF driver:

7.4.2.2.1 Using the JPPFDriver API

This can be done by calling the method [getJobTasksListenerManager\(\)](#) on the [JPPFDriver](#) instance provided by another driver plugin, add-on or extension. For instance, as in this [driver startup class](#):

```
public class MyDriverStartup implements JPPFDriverStartupSPI {
    JPPFDriver driver;

    @Override
    public void run() {
        // get the object which manages task return listeners
        JobTasksListenerManager manager = this.driver.getJobTasksListenerManager();
        // register a new TaskReturnListener
        manager.addJobTasksListener(new JobTasksListener() {
            @Override
            public void tasksDispatched(JobTasksEvent event) { ... }
            @Override
            public void tasksReturned(JobTasksEvent event) { ... }
            @Override
            public void resultsReceived(JobTasksEvent event) { ... }
        });
    }

    // JPPF will call this method before run()
    public void setDriver(JPPFDriver driver) {
        this.driver = driver;
    }
}
```

Note that you will need to have the jppf-server.jar file in your build path for this code to compile.

7.4.2.2.2 Using the Service Provider Interface (SPI)

To register the [JobTasksListener](#) implementation as a standalone service, create a service definition file named **"org.jppf.job.JobTasksListener"** in the **"META-INF/services"** directory. In this file, add, for each of your listener implementations, a line containing the fully qualified name of the implementation class. For instance, if we defined two implementations `MyJobTasksListener1` and `MyJobTasksListener2` in the `test1` and `test2` packages, respectively, then the service definition file should contain:

```
test1.MyJobTasksListener1
test2.MyJobTasksListener2
```

Note that, to work with the SPI, each implementation *must* have a no-args constructor, whether implicit or explicit.

7.4.3 Receiving server statistics events

Reminder: if you are not familiar with the server statistics API, please refer to the section "[The JPPF statistics API](#)" of this user guide.

7.4.3.1 Implementation

To receive notifications of changes in the server statistics, you will need to create a class with a no-arg constructor which extends [JPPFFilteredStatisticsListener](#), which is defined as follows:

```
public abstract class JPPFFilteredStatisticsListener implements JPPFStatisticsListener {
    // Get an optional filter to associate with this listener
    public JPPFStatistics.Filter getFilter()
}
```

The method `getFilter()` returns a [JPPFStatistics.Filter](#) which may be `null`, and is defined as:

```
public class JPPFStatistics implements Serializable, Iterable<JPPFSnapshot> {
    // A filter interface for snapshots
    public interface Filter {
        // Determines whether the specified snapshot is accepted by this filter
        boolean accept(JPPFSnapshot snapshot);
    }
}
```

The statistics listener will only receive events for the snapshots accepted by the filter. If it is `null`, then it will receive events for all the snapshots.

[JPPFFilteredStatisticsListener](#) implements the [JPPFStatisticsListener](#) interface, which provides the following event notification methods:

```
public interface JPPFStatisticsListener extends EventListener {
    // Called when a new snapshot is created
    void snapshotAdded(JPPFStatisticsEvent event);

    // Called when a snapshot is removed
    void snapshotRemoved(JPPFStatisticsEvent event);

    // Called when a snapshot is updated
    void snapshotUpdated(JPPFStatisticsEvent event);
}
```

Each [JPPFStatisticEvent](#) object provides the following information:

```
public class JPPFStatisticsEvent extends EventObject {
    // Get the statistics source of this event
    public JPPFStatistics getStatistics()

    // Get the snapshot that was created, removed or updated
    public JPPFSnapshot getSnapshot()
}
```

Note: since updates to the statistics in the server have a very high frequency, it is recommended to make the listener methods as short as possible, with regards to their execution time. This will avoid a significant impact to the server performance. If you can't avoid it, then you may consider posting the events into a queue for asynchronous processing in a separate thread.

7.4.3.2 Deployment

The deployment of a statistics is done via the Service Provider Interface (SPI):

- create a file named "**org.jppf.utils.stats.JPPFFilteredStatisticsListener**" in the **META-INF/services** folder
- in this file add a line containing the fully qualified name of your listener implementation class, for instance "mypackage.MyClass"
- include the jar file or class folder containing both your implementation and service in the server's classpath

7.4.3.3 Full example

Let's write a statistics listener which prints all the events it receives to the output console:

```
package test.stats;

import org.jppf.utils.stats.*;

public class MyStatisticsListener extends JPPFFilteredStatisticsListener {
    public MyStatisticsListener() {
        System.out.println("creating new statistics listener");
    }

    @Override
    public void snapshotAdded(JPPFStatisticsEvent event) {
        System.out.printf("added '%s'%n", event.getSnapshot().getLabel());
    }

    @Override
    public void snapshotRemoved(JPPFStatisticsEvent event) {
        System.out.printf("removed '%s'%n", event.getSnapshot().getLabel());
    }

    @Override
    public void snapshotUpdated(JPPFStatisticsEvent event) {
        System.out.printf("updated '%s'%n", event.getSnapshot().getLabel());
    }

    @Override
    public JPPFStatistics.Filter getFilter() {
        System.out.println("in MyStatisticsListener.getFilter()");
        return super.getFilter();
    }
}
```

Now, for the server to register this listener, we create the file:

"META-INF/services/org.jppf.utils.stats.JPPFFilteredStatisticsListener"
with the following single line: **"test.stats.MyStatisticsListener"**

Our listener is now ready to be deployed.

7.4.4 Custom discovery of peer drivers

7.4.4.1 Rationale

The current built-in peer driver discovery mechanisms in the JPPF server have potential shortcomings:

- automatic discovery relies on the UDP multicast protocol, which is not available in all environments. For instance, cloud environments do not allow it.
- the manual configuration of the connections requires that all possible peer drivers be known in advance, which prevents new, previously unknown drivers from being discovered

To overcome these limitations, and to allow more flexibility as to how peer drivers are discovered, you can now implement your own custom discovery mechanism.

7.4.4.2 Implementation

A driver discovery mechanism is implemented by extending the class [PeerDriverDiscovery](#), defined as:

```
public abstract class PeerDriverDiscovery extends DriverDiscovery<DriverConnectionInfo> {  
}
```

As we can see, this class has no method of its own, and the interesting methods are in its superclass [DriverDiscovery](#):

```
public abstract class DriverDiscovery<E extends DriverConnectionInfo> {  
    // Perform the driver discovery. This method runs in its own separate thread  
    public abstract void discover() throws InterruptedException;  
  
    // Notify that a new driver was discovered  
    protected void newConnection(E info)  
  
    // Shut this discovery down. This method is intended to be overridden  
    // in subclasses to allow user-defined cleanup operations  
    public void shutdown()  
}
```

By default, the `shutdown()` method does nothing and is intended to be overridden in subclasses if needed. It is called when the JPPF driver is shut down and is thus part of the discovery's life cycle.

The actual discovery is performed within the `discover()` method. From this method, for each discovered driver you must call the `newConnection()` method, passing an instance of [DriverConnectionInfo](#) defined as:

```
public class DriverConnectionInfo  
{  
    // Get the name given to this connection  
    public String getName()  
    // Determine whether secure (with SSL/TLS) connections should be established  
    public boolean isSecure()  
    // Get the driver host name or IP address  
    public String getHost()  
    // Get the driver port to connect to  
    public int getPort()  
    // Get the number of connections (pool size) to establish with the peer driver  
    public int getPoolSize()  
}
```

Here is an example of a very simple implementation:

```
public class SimplePeerDiscovery extends PeerDriverDiscovery {  
    @Override  
    public void discover() throws InterruptedException {  
        // new peer connection named "myPeerDriver" with a pool size of 2  
        newConnection(  
            new DriverConnectionInfo("myPeerDriver", false, "www.myhost.org", 11111, 2));  
    }  
}
```

The `discover()` method of a [PeerDriverDiscovery](#) is not limited to calling `newConnection()` only once. You can invoke this method with as many instances of [DriverConnectionInfo](#) as you wish, allowing for the discovery of as many peer drivers as required.

Additionally, the `discover()` method runs in its own separate thread and doesn't have to terminate immediately. This means that you can run a loop inside that may, for instance, perform a periodic polling of an (external) service to discover new peer drivers over time. Here is a more complex example illustrating this:

```
public class PollingDiscovery extends PeerDriverDiscovery {
    // whether this discovery was shutdown
    private boolean shutdownFlag = false;

    @Override
    public void discover() throws InterruptedException {
        while (!isShutdown()) {
            List<DriverConnectionInfo> peers = externalLookup();
            if (drivers != null) {
                for (DriverConnectionInfo peer: peers) newConnection(peer);
            }
            synchronized(this) {
                // wait 5 seconds before the next lookup
                wait(5000L);
            }
        }
    }

    // Query an external service for discovered drivers
    public List<DriverConnectionInfo> externalLookup() {
        return ...;
    }

    public synchronized boolean isShutdown() {
        return shutdownFlag;
    }

    @Override
    public synchronized void shutdown() {
        shutdownFlag = false;
        // wake up the discover() thread
        notify();
    }
}
```

7.4.4.3 Deployment via SPI

A driver discovery mechanism can be automatically loaded and installed via the Service Provider Interface (SPI):

- in your source or resources folder, create a file **META-INF/services/org.jppf.discovery.PeerDriverDiscovery**
- edit this file and add a line with the fully qualified class name of your implementation of [PeerDriverDiscovery](#), for instance `org.jppf.example.SimplePeerDiscovery`.
- make sure that this file, along with the implementation class, is in the classpath of your server

7.4.4.4 Deployment via API

The class [JPPFDriver](#) provides an API to add or remove [PeerDriverDiscovery](#) implementations at any time:

```
public class JPPFDriver {
    // Add a custom driver discovery mechanism to those already registered, if any
    public void addDriverDiscovery(PeerDriverDiscovery discovery)

    // Remove a custom driver discovery mechanism from those already registered
    public void removeDriverDiscovery(PeerDriverDiscovery discovery)
}
```

To obtain an instance of [JPPFDriver](#), you will need to wrap the code in a driver extension that provides it, for instance a [driver startup class](#):

```

public class MyDiscoveryInitializer implements JPPFDriverStartupSPI {
    JPPFDriver driver;

    @Override
    public void run() {
        // create and add the peer discovery provider
        driver.addDriverDiscovery(new SimplePeerDiscovery());
    }

    // JPPF will call this method before run()
    public void setDriver(JPPFDriver driver) {
        this.driver = driver;
    }
}

```

7.4.4.5 Tip

To prevent the built-in peer driver discovery mechanisms from starting, set the following driver configuration properties:

```

jppf.peer.discovery.enabled = false
# set an empty value or simply comment it out
jppf.peers =

```

This way, only custom peer driver discovery mechanisms will be used.

7.4.5 Implementing a custom job persistence

For full details on how the jobs persistence in the driver works, we invite you to read the [dedicated documentation chapter](#).

7.4.5.1 Implementation

The jobs persistence facility relies on one or more implementations of the [JobPersistence](#) interface, defined as follows:

```
public interface JobPersistence {
    // Store the specified job elements. All elements are part of the same job
    void store(Collection<PersistenceInfo> infos) throws JobPersistenceException;

    // Load the specified job elements. All elements are part of the same job
    List<InputStream> load(Collection<PersistenceInfo> infos)
        throws JobPersistenceException;

    // Get the UUIDs of all persisted job
    List<String> getPersistedJobUuids() throws JobPersistenceException;

    // Get the positions of all the tasks in the specified job
    int[] getTaskPositions(String jobUuid) throws JobPersistenceException;

    // Get the positions of all the task results in the specified job
    int[] getTaskResultPositions(String jobUuid) throws JobPersistenceException;

    // Delete the persisted job with the specified UUID
    void deleteJob(String jobUuid) throws JobPersistenceException;

    // Determine whether a job is persisted, that is, present in the persistence store
    boolean isJobPersisted(String jobUuid) throws JobPersistenceException;
}
```

Note that all the methods in this interface define callbacks that are called by the JPPF driver during the life cycle of the jobs it processes.

If the persistence implementation does not require any parameter to be passed on from the configuration, then it can simply use a default no-arg constructor, implicit or not. Otherwise, string parameters can be specified in the configuration and can be passed on to the implementation via either:

- a public constructor that takes a `String... vararg` parameter
- a public void `setParameter(String...)` method

If both are defined, then JPPF will always use the constructor.

The `store()` and `load()` methods both take a collection of objects of type [PersistenceInfo](#), defined as follows:

```
public interface PersistenceInfo extends Serializable {
    // Get the job uuid
    String getJobUuid();

    // Get the related job information
    JPPFDistributedJob getJob();

    // Get the type of persisted object
    PersistenceObjectType getType();

    // Get the position of the task or task result in the job
    int getPosition();

    // Get an input stream for the persisted object
    InputStream getInputStream() throws Exception;

    // Get the size in bytes of the persisted object
    int getSize();
}
```

Each instance of this class represents a job element, which can be either a job header, a data provider, a task or a task result. The type of job element can be obtained via the `getType()` method, which returns a [PersistenceObjectType](#) enum element.

Additionally, the fact that `store()` and `load()` take a collection of [PersistenceInfo](#) objects means that multiple job elements can be persisted in the same transaction, in the case of a transactional persistence implementation.

You will also notice that the job element's data is only available via an input stream: the persisted data represents a serialized object graph which can be temporarily stored anywhere. Here, the serialization format is completely transparent to the job persistence implementation.

Note: the `load()` method returns a list of `InputStream` objects. Each `InputStream` corresponds to a [PersistenceInfo](#) object passed in the input collection. The list's ordering **must** be the same as the ordering defined by the input collection's iterator.

7.4.5.2 Configuration

Configuring a job persistence implementation is done via the "**jppf.job.persistence**" configuration property, by passing it the fully qualified name of the implementation class, along with its optional parameters:

```
jppf.job.persistence = <full_class_name> <param1> ... <paramN>
```

for instance, if we define the following implementation:

```
package test.persistence;
...
public class MyPersistence implements JobPersistence {
    public MyPersistence(String...params) {
        String dataSourceName = (params.length > 0) ? params[0] : "myDS";
        int numberOfThreads = (params.length > 1) ? Integer.valueOf(params[1]) : 1;
        // ... initialize ...
    }
    ... implementation of JobPersistence ...
}
```

then we could configure it as follows:

```
jppf.job.persistence = test.persistence.MyPersistence testDS 4
```

7.4.5.3 Examples

Implementing a custom job persistence is a non-trivial task. Rather than just listing the code of a lengthy real-life implementation, or a simplified implementation that would be unrealistic, we prefer to provide links to the code of the predefined built-in implementations:

- all built-in implementations are in the [org.jppf.job.persistence.impl](#) package
- default database persistence: class [DefaultDatabasePersistence](#) ([javadoc](#))
- default file persistence: class [DefaultFilePersistence](#) ([javadoc](#))
- asynchronous persistence wrapper: class [AsynchronousPersistence](#) ([javadoc](#))
- cacheable persistence wrapper: class [CacheablePersistence](#) ([javadoc](#))

Note: the source code of these classes is also available from within the Javadoc.

7.5 Nodes

7.5.1 Receiving node life cycle events

This plugin provides the ability to receive notifications of major events occurring within a node, including node startup and termination as well as the start and completion of each job processing.

7.5.1.1 *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);

    // Called after the node has sent the results of a job to the server and before it
    // receives the next job. The node is fully idle at this point.
    void beforeNextJob(NodeLifeCycleEvent event);
}
```

If you do not wish to implement all the methods of this interface you may instead extend the convenience class [NodeLifeCycleListenerAdapter](#), which provides an empty implementation of all the methods of the interface, and override only the methods you're interested in.

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,

    // jobEnding() notification
    JOB_ENDING,

    // beforeNextJob() notification
    BEFORE_NEXT_JOB
}

```

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 [AbstractTask](#) for all practical purposes.

[JPPFDistributedJob](#) is an interface common to client side jobs (see [JPPFJob](#)) and server / node side jobs. It provides the following methods, which can be used in 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.getTaskClassLoader()`, 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 extends NodeLifeCycleListenerAdapter {
    @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.getName() +
        "' using task class loader " + event.getTaskClassLoader());
}

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

@Override
public void jobEnding(NodeLifeCycleEvent event) {
    System.out.println("node finished job '" + event.getJob().getName() + "'");
}
}

```

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](#)

7.5.1.2 Error handler

It is now possible to provide an error handler for each `NodeLifeCycleListener` implementation. This error handler will process all uncaught `Throwables` 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

7.5.2 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 [InitializationHook](#), which is defined as follows:

```
public interface InitializationHook {  
    // Called each time the node is about to attempt to connect to a driver  
    void initializing(TypedProperties 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 [TypedProperties](#), 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  
    @Override  
    public void initializing(TypedProperties initialConfiguration) {  
        // store the alternate server address  
        if (alternateServer == null) {  
            alternateServer = initialConfiguration.getString("alternate.server.host");  
        }  
        // 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");  
            initialConfiguration.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  
            initialConfiguration.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](#).

7.5.3 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.execute.ThreadManagerForkJoin
```

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

```
public class WordCountTask extends JPPFTask {
    // a list of documents 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:

```
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;
    }

    // return a mapping of each word to its number of occurrences
    @Override
    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: [Fibonacci Fork/Join sample..](#)

7.5.4 Receiving 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 notifications 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.classloader.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.classloader.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.

7.5.5 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 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.fireNotification(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()

    // Whether this is a user-defined event sent from a task
    public boolean isUserNotification();
}
```

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 listener 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.

7.5.6 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.

7.5.6.1 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. Additionally, 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
    }

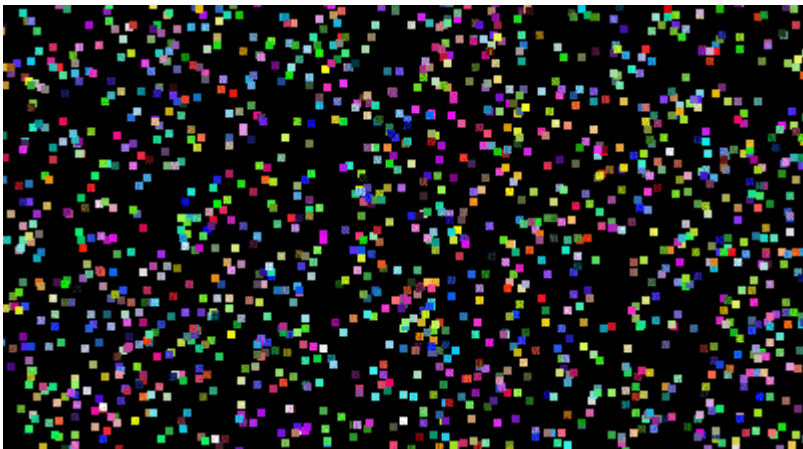
    @Override
    public void destroy() {
        if (timer != null) timer.cancel();
    }
}
```

```

@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:



7.5.6.2 Integrating with node events

To receive notifications of node life 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:

```

// 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;
    }
}

```

7.5.6.3 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

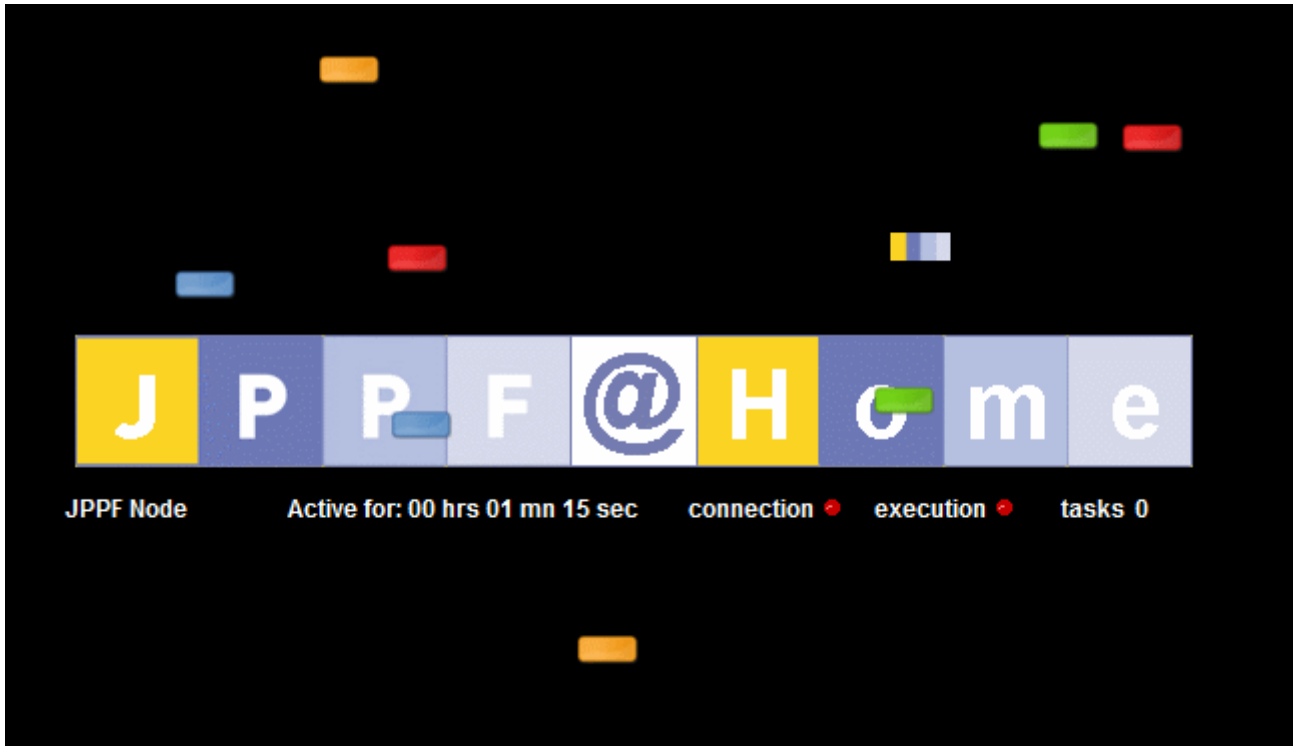
# width and height (in pixels), only apply if fullscreen = false. Default to 1000x800
jppf.screensaver.width = 1000
jppf.screensaver.height = 800

# close on mouse motion in full screen mode? default to true
jppf.screensaver.mouse.motion.close = true

```

7.5.6.4 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 specific 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
```

7.5.7 Defining the node connection strategy

By default, JPPF nodes rely on their configuration to find out the information required to connect to a server: either via UDP multicast discovery when discovery is enabled, or via manually set configuration properties for the server host, port, whether SSL is enabled, etc... This makes it potentially complex to define what the behavior should be when the node fails to connect to a server. For example this does not allow to define which server(s) a node should fail over to in case a server dies or is no longer reachable via the network.

The connection strategy add-on provides a simple way to specify which server a node should connect to and how to react when it fails to do so.

7.5.7.1 The *DriverConnectionStrategy* interface

The node's strategy to connect to a driver is defined as an implementation of the interface [DriverConnectionStrategy](#), which is defined as follows:

```
// Defines which parameters should be used to connect to the driver
public interface DriverConnectionStrategy {
    // Get a new connection information, eventually based on the current one
    DriverConnectionInfo nextConnectionInfo(
        DriverConnectionInfo currentInfo, ConnectionContext context);
}
```

This interface defines a single method which takes two parameters as input, and returns a [DriverConnectionInfo](#) object, which encapsulates the information required to connect to a JPPF driver..

The first parameter represents the current connection information, that is, the information that was used for the last connection attempt. When the node connects for the first time, this parameter will be null. This parameter is an instance of the interface [DriverConnectionInfo](#), defined as follows:

```
public interface DriverConnectionInfo {
    // determine whether secure SSL/TLS connections should be established
    boolean isSecure();

    // get the driver host name or IP address
    String getHost();

    // get the driver port to connect to
    int getPort();

    // whether the heartbeat (recovery) mechanism is enabled
    boolean isRecoveryEnabled();
}
```

JPPF provides a ready-to-use implementation of this interface with the class [JPPFDriverConnectionInfo](#).

The second parameter represents the context of the connection or reconnection request, basically explaining why the request is made. It could be due to a management request or to an error occurring within the node, or simply the first connection attempt at node startup time. It is designed to help make a decision about which driver to connect to. It is an instance of the class [ConnectionContext](#), defined as:

```
public class ConnectionContext {
    // get an explanation text for the reconnection
    public String getMessage()

    // get an eventual Throwable that triggered the reconnection
    public Throwable getThrowable()

    // get the reason for the connection or reconnection
    public ConnectionReason getReason()
}
```


The `getReason()` method returns a reason code among those defined in the [ConnectionReason](#) enum:

```
public enum ConnectionReason {
    // indicates the first connection attempt when the node starts up
    INITIAL_CONNECTION_REQUEST,
    // a reconnection was requested via the management APIs or admin console
    MANAGEMENT_REQUEST,
    // An error occurred while initializing the class loader connection
    CLASSLOADER_INIT_ERROR,
    // an error occurred while processing a class loader request
    CLASSLOADER_PROCESSING_ERROR,
    // an error occurred during the job channel initialization
    JOB_CHANNEL_INIT_ERROR,
    // an error occurred on the job channel while processing a job
    JOB_CHANNEL_PROCESSING_ERROR,
    // the heartbeat mechanism failed to receive a message from the server
    HEARTBEAT_FAILURE
}
```

The following example implementation uses a set of driver connection information objects stored in a queue and performs a round-robin selection at each connection request:

```
public class MyConnectionStrategy implements DriverConnectionStrategy {
    // the queue in which DriverConnectionInfo objects are stored
    private final Queue<DriverConnectionInfo> queue = new LinkedBlockingQueue<>();

    // initialize the set of drivers to connect to
    public MyConnectionStrategy() {
        queue.offer(new JPPFDriverConnectionInfo(false, "192.168.1.11", 11111, false));
        queue.offer(new JPPFDriverConnectionInfo(false, "192.168.1.12", 11111, false));
        queue.offer(new JPPFDriverConnectionInfo(true, "192.168.1.13", 11443, false));
    }

    @Override
    public DriverConnectionInfo nextConnectionInfo(
        DriverConnectionInfo currentInfo, ConnectionContext context) {
        DriverConnectionInfo info;
        // if the reconnection is requested via management, keep the current driver info
        if ((currentInfo != null) &&
            (context.getReason() == ConnectionReason.MANAGEMENT_REQUEST)) {
            info = currentInfo;
        } else {
            // extract the next info from the queue
            info = queue.poll();
            // put it back at the end of the queue
            queue.offer(info);
        }
        return info;
    }
}
```

Note: JPPF can also provide access to the node configuration, by passing it to a constructor that takes a single [TypedProperties](#) parameter, as in this example:

```
public class MyConnectionStrategy implements DriverConnectionStrategy {
    // the node configuration passed to the constructor
    private final TypedProperties configuration;

    public MyConnectionStrategy(TypedProperties configuration) {
        this.configuration = configuration;
    }

    @Override
    public DriverConnectionInfo nextConnectionInfo(
        DriverConnectionInfo currentInfo, ConnectionContext context) {
        ...
    }
}
```

7.5.7.2 Plugging the strategy into the node

Specifying which connection strategy a node should use is done in the node's configuration as follows:

```
# fully qualified name of a class implementing DriverConnectionStrategy
```

```
jppf.server.connection.strategy = test.MyConnectionStrategy
```

As stated in the comment, the value of the “jppf.server.connection.strategy” property is the fully qualified name of a class implementing [DriverConnectionStrategy](#), which must also have a no-args constructor. If this property is left unspecified, or if the specified class cannot be instantiated, the node will default to an instance of [JPPFDefaultConnectionStrategy](#), which uses the configuration to find out the connection information, either via server discovery or from the manually specified server-related properties.

7.5.7.3 Built-in strategies

7.5.7.3.1 File-based CSV server definitions

In addition to the default [JPPFDefaultConnectionStrategy](#), JPPF provides a connection strategy which reads a list of driver connection information from a CSV file. As in the example above, it will perform a round-robin selection of the drivers to connect to. Additionally, if the specified CSV file is invalid or cannot be read, or none of its entries is valid, it will default to [JPPFDefaultConnectionStrategy](#).

This implementation is named [JPPFCsvFileConnectionStrategy](#) and is configured as follows:

```
# read the driver connection info from a CSV file
jppf.server.connection.strategy = org.jppf.node.connection.JPPFCsvFileConnectionStrategy
# location of the CSV file
jppf.server.connection.strategy.file = /home/me/data/drivers.csv
```

The file is first looked up in the file system at the specified location, then in the classpath if not found in the file system.

The syntax and format for the entries in the CSV files are as in the following example:

```
# CSV columns: ssl_enabled, server_host, server_port, recovery_enabled

# server 1
false, 192.168.1.15, 11111, false
# server 2, with recovery enabled
false, 192.168.1.16, 11111, true
# server 3, with SSL enabled
true, 192.168.1.17, 11443, false
```

Please note that any line starting with a '#' (after trimming) is considered a comment.

7.5.7.3.2 Configuration-based CSV server definitions

[JPPFCsvPropertyConnectionStrategy](#) is very similar to [JPPFCsvFileConnectionStrategy](#), with the difference that it takes the values from a single configuration property instead of from a file, with the following format:

```
jppf.server.connection.strategy.definitions = \
secure1, host1, port1, recovery_enabled1 | \
... | \
secureN, hostN, portN, recovery_enabledN
```

where:

- each connection definition is represented as a group of comma-separated values
- csv groups are separated with the '|' (pipe) character
- in each csv group:
 - *secure_i* is a boolean value (either 'true' or 'false', case-insensitive) indicating whether SSL/TLS connectivity is enabled. Any value that is not 'true' is interpreted as 'false'.
 - *host_i* is the host name or ip address of the driver to connect to
 - *port_i* is the port to connect to on the driver host
 - *recovery_enabled_i* determines whether the recovery (heartbeat) mechanism should be enabled for the node
 - if the first character is a '#' then the group is considered a comment and ignored.

Here is an example configuration:

```
# read the driver connection info from a configuration property
jppf.server.connection.strategy = \
    org.jppf.node.connection.JPPFCsvPropertyConnectionStrategy

# definitions of the server connections
jppf.server.connection.strategy.definitions = \
    # definition for server 1 |\
    false, my.host1.org, 11111, false |\
    # definition for server 2 |\
    true, my.host2.org, 11443, true
```

Note the multiline syntax that makes the configuration easier to read.

7.5.8 Node throttling

7.5.8.1 Definition and rationale

This pluggable mechanism allows a JPPF node to warn a driver that it can no longer accept jobs, whenever a condition is reached. When the condition no longer applies, a new notification is sent to the driver to specify that the node accepts jobs again.

The main purpose of this facility is to avoid resource exhaustion in the nodes, since by default a node can process an unlimited number of jobs at any given time. However, it can be used for other purposes, for instance to define time windows during which a node will or will not accept work.

7.5.8.2 How it works

The node throttling relies on the service provider interface (SPI) mechanism. The service interface, which is the contract between the node and a plugin implementation, is represented by the [JPPFNodeThrottling](#) interface, defined as:

```
public interface JPPFNodeThrottling {  
    // Determine whether the node accepts new jobs  
    boolean acceptsNewJobs(Node node);  
}
```

An implementation of [JPPFNodeThrottling](#) will be invoked at two points in the node's life cycle:

- during the initial handshake with the driver
- at regular intervals via a dedicated timer

The duration of the interval between periodic checks is configurable in the node with the following property, expressed in milliseconds and defaulting to 2000 milliseconds:

```
# interval between throttling checks in milliseconds  
jppf.node.throttling.check.period = 2000
```

Note 1: You can define and deploy any number of throttling plugin implementations in a node.

Note 2: the throttling mechanism does not apply to [offline nodes](#), since offline nodes only accept a single job at a time.

7.5.8.3 Implementing a node throttling plugin

Let's take a simple example implementation which causes the node to no longer accept new jobs when the heap usage is more than 90% of the maximum heap size:

```
public class HeapBasedThrottling implements JPPFNodeThrottling {  
    @Override  
    public boolean acceptsNewJobs(final Node node) {  
        final double pct = getUsedMemoryPct();  
        final boolean acceptsJobs = pct < 90d;  
        // invoke a GC so that at next check the node may be accepting jobs again  
        if (!acceptsJobs) {  
            System.gc();  
        }  
        return acceptsJobs;  
    }  
  
    // Compute the percentage of currently used heap memory  
    // where used heap ratio = (current heap size - current free heap) / max heap size  
    private static double getUsedMemoryPct() {  
        final Runtime rt = Runtime.getRuntime();  
        final double usedMemory = rt.totalMemory() - rt.freeMemory();  
        return 100d * usedMemory / rt.maxMemory();  
    }  
}
```

7.5.8.4 Deploying the plugin

As usual with an SPI-based service, follow these steps to deploy:

- create a service file in **META-INF/services** named “**org.jppf.node.throttling.JPPFNodeThrottling**”
- in this file, add the fully qualified class name of your implementation of the [JPPFNodeThrottling](#) 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 plugin.

7.5.8.5 Built-in implementation: heap usage-based throttling

The class [MemoryThresholdThrottling](#) is a slightly more sophisticated version of the example we have seen above. It is also a throttling mechanism that causes a node to not accept jobs whenever heap usage reaches a configured percentage of the maximum heap size. It can be configured with the following properties:

- it is deactivated by default and can be activated by setting the property (false by default):

```
jppf.node.throttling.memory.threshold.active = true
```

- the threshold can be set as a percentage of the maximum heap size with this property (90% by default):

```
jppf.node.throttling.memory.threshold = 87.5
```

- it can also call `System.gc()`, in an attempt to mitigate the high heap usage, after `acceptsNewJobs(Node)` returns false a configured consecutive number of times. The corresponding configuration property is set as (3 by default):

```
jppf.node.throttling.memory.threshold.maxNbTimesFalse = 2
```

Example configuration:

```
# activate the throttling plugin
jppf.node.throttling.memory.threshold.active = true
# % of maximum heap size that causes the node to refuse jobs
jppf.node.throttling.memory.threshold = 87.5
# number of time the check returns false before invoking System.gc()
jppf.node.throttling.memory.threshold.maxNbTimesFalse = 5
```

7.6 Clients

7.6.1 Custom discovery of remote drivers

7.6.1.1 Rationale

The current built-in driver discovery mechanisms in the JPPF client have potential shortcomings:

- automatic discovery relies on the UDP multicast protocol, which is not available in all environments. For instance, cloud environments do not allow it.
- the manual configuration of the connections requires that all possible drivers be known in advance, which prevents new, previously unknown drivers from being discovered

To overcome these limitations, and to allow more flexibility as to how drivers are discovered by a client, you can now implement your own custom discovery mechanism.

7.6.1.2 Implementation

A driver discovery mechanism is implemented by extending the class [ClientDriverDiscovery](#), defined as:

```
public abstract class ClientDriverDiscovery
    extends DriverDiscovery<ClientConnectionPoolInfo> {
}
```

As we can see, this class has no method of its own, and the interesting methods are in its superclass [DriverDiscovery](#):

```
public abstract class DriverDiscovery<E extends DriverConnectionInfo> {
    // Perform the driver discovery. This method runs in its own separate thread
    public abstract void discover() throws InterruptedException;
    // Notify that a new driver was discovered
    protected void newConnection(E info)
    // Shut this discovery down. This method is intended to be overridden
    // in subclasses to allow user-defined cleanup operations
    public void shutdown()
}
```

By default, the `shutdown()` method does nothing and is intended to be overridden in subclasses if needed. It is called when closing the JPPF client and is thus part of the discovery's life cycle.

The actual discovery is performed within the `discover()` method. From this method, for each discovered driver you must call the `newConnection()` method, passing an instance of [ClientConnectionPoolInfo](#), defined as follows:

```
public class ClientConnectionPoolInfo extends DriverConnectionInfo
    // Initialize a pool of plain connections with default name("driver"),
    // host ("localhost"), port (11111), priority (0) and pool sizes (1)
    public ClientConnectionPoolInfo()
    // Initialize a pool of plain connections with default priority (0) and pool sizes (1)
    public ClientConnectionPoolInfo(String name, String host, int port)
    // Initialize a pool of connections with default priority (0) and pool sizes (1)
    public ClientConnectionPoolInfo(String name, boolean secure, String host, int port)
    // Initialize a pool of connections with the specified parameters
    public ClientConnectionPoolInfo(String name, boolean secure, String host, int port,
        int priority, int poolSize, int jmxPoolSize)
    // Initialize a pool of connections with the specified parameters
    public ClientConnectionPoolInfo(String name, boolean secure, String host, int port,
        int priority, int poolSize, int jmxPoolSize, int maxJobs)

    // Get the connection priority
    public int getPriority()
    // Get the connection pool size
    public int getPoolSize()
    // Get the associated JMX connection pool size
    public int getJmxPoolSize()
    // Get the number of jobs that can be handled concurrently by each connection
    public int getMaxJobs()
}
```

The super class [DriverConnectionInfo](#) provides getters for the name, host, port and secure attributes:

```
public class DriverConnectionInfo
// Get the name given to this connection
public String getName()

// Determine whether secure (with SSL/TLS) connections should be established
public boolean isSecure()

// Get the driver host name or IP address
public String getHost()

// Get the driver port to connect to
public int getPort()
}
```

Here is an example of a very simple implementation:

```
public class SimpleDiscovery extends ClientDriverDiscovery {
@Override
public void discover() throws InterruptedException {
// new connection pool named "myDriver"
newConnection(
new ClientConnectionPoolInfo("myDriver", false, "www.myhost.org", 11111));
}
}
```

The `discover()` method of a [ClientDriverDiscovery](#) is not limited to calling `newConnection()` only once. You can invoke this method with as many instances of [ClientConnectionPoolInfo](#) as you wish, allowing for the discovery of as many connection pools as required.

Additionally, the `discover()` method runs in its own separate thread and doesn't have to terminate immediately. This means that you can run a loop inside that may, for instance, perform a periodic polling of an (external) service to discover new drivers over time. Here is a more complex example illustrating this:

```
public class LoopingDiscovery extends ClientDriverDiscovery {
// whether this discovery was shutdown
private boolean shutdownFlag = false;

@Override
public void discover() throws InterruptedException {
while (!isShutdown()) {
List<ClientConnectionPoolInfo> drivers = externalLookup();
if (drivers != null) {
for (ClientConnectionPoolInfo driver: drivers) newConnection(driver);
}
synchronized(this) { // wait 5 seconds before the next lookup
wait(5000L);
}
}
}

// Query an external service for discovered drivers
public List<ClientConnectionPoolInfo> externalLookup() {
return ...;
}

public synchronized boolean isShutdown() {
return shutdownFlag;
}

@Override
public synchronized void shutdown() {
shutdownFlag = true;
notify(); // wake up the discover() thread
}
}
```

7.6.1.3 Deployment via SPI

A driver discovery mechanism can be automatically loaded and installed via the Service Provider Interface (SPI):

- in your source or resources folder, create a file **META-INF/services/org.jpmp.discovery.ClientDriverDiscovery**

- edit this file and add a line with the fully qualified class name of your subclass of [ClientDriverDiscovery](#), for instance `org.jppf.example.SimpleDiscovery`.
- make sure that this file, along with the implementation class, is in the classpath of your client application

7.6.1.4 Deployment via API

The class [JPPFClient](#) provides an API to add or remove [ClientDriverDiscovery](#) implementations at any time:

```
public class JPPFClient extends AbstractGenericClient {  
    // Add a custom driver discovery mechanism to those already registered, if any  
    public void addDriverDiscovery(ClientDriverDiscovery discovery)  
  
    // Remove a custom driver discovery mechanism from those already registered  
    public void removeDriverDiscovery(ClientDriverDiscovery discovery)  
}
```

Example usage:

```
JPPFClient client = new JPPFClient();  
ClientDriverDiscovery discovery = new SimpleDiscovery();  
client.addDriverDiscovery(discovery);
```

7.6.1.5 Tip

To prevent the built-in discovery mechanisms from starting, set the following client configuration property:

in the configuration file:

```
jppf.remote.execution.enabled = false
```

or via the configuration API:

```
JPPFConfiguration.set(JPPFProperties.REMOTE_EXECUTION_ENABLED, false);
```

This way, only custom driver discovery mechanisms will be used.

7.7 Administration console

7.7.1 Embedding the administration console

As of version 5,0, JPPF provides a very simple API to obtain a reference to the Swing component which encloses the administration console user interface, and add it to an existing Swing-based UI. This reference is obtained by calling the static method [getAdminConsole\(\)](#) of the class [JPPFAdminConsole](#), which returns a [JComponent](#) object. This JComponent can then be added to any Swing container.

The JPPF admin console will only be created once, even if `JPPFAdminConsole.getAdminConsole()` is called multiple times. In other words, it is guaranteed that the same JComponent instance will always be returned.

[JPPFAdminConsole](#) also provides access to the [JPPF client](#), [topology manager](#) and [job monitor](#) that the console uses for its monitoring and management operations:

```
public class JPPFAdminConsole {
    // Get the admin console as a JComponent that can be added to a Swing container
    public static JComponent getAdminConsole();

    // Get the JPPF client used by the admin console
    public static JPPFClient getJPPFClient();

    // Get the topology manager used by the admin console
    public static TopologyManager getTopologyManager();

    // Get the job monitor used by the admin console
    public static JobMonitor getJobMonitor();

    // Launch the administration console as a standalone application
    public static void main(final String[] args);
}
```

Here is an example of how it can be used:

```
import javax.swing.*;

// these imports require jppf-admin.jar
import org.jppf.ui.console.JPPFAdminConsole;
import org.jppf.ui.utils.GuiUtils;

public class EmbeddedConsole {
    public static void main(final String[] args) throws Exception {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        JPanel mainPanel = new JPanel();
        // layout the components vertically within a box
        mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.Y_AXIS));
        // the top component is a label with an image and some text
        ImageIcon icon = GuiUtils.loadIcon("../admin/jppf_logo.gif");
        JLabel label = new JLabel("Test JPPF embedded console", icon, SwingConstants.LEFT);
        mainPanel.add(label);
        // add the JPPF admin console as the bottom component
        mainPanel.add(JPPFAdminConsole.getAdminConsole());
        // finally, add the UI to a JFrame and display it
        JFrame frame = new JFrame("Embedded console");
        frame.setSize(800, 600);
        frame.add(mainPanel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Since [JPPFAdminConsole](#) is part of the console code, you will need to include "jppf-admin.jar" in your build path or class path, both at compile time and at runtime.

When embedded within an external GUI, the admin console will probably look different than when run as a standalone application. This is due to the Swing look and feel that is used. When run standalone, the console uses the [JGoodies Looks](#) v2.2.2 "PlasticXPLookAndFeel".

The GUI resulting from the example code above will look like this:

Embedded console

J P P F Test JPPF embedded console

Server connection 192.168.1.24:11192

Topology Job Data Node Data Charts Load Balancing

Tree view Graph view JVM health

Node Threads... Node Status Exec status Tasks Executed Nb slave nodes

192.168.1.24:11192	127.0.0.1:12002	8 / 5	Connected	Idle	0	0
192.168.1.24:11191	192.168.1.24:11192					
192.168.1.24:11192	192.168.1.24:12001	8 / 5	Connected	Idle	0	0

Enter the number of threads and...

Number of threads 8

Threads priority (1 - 10) 5

OK Cancel

Active servers 2 Active nodes 2

7.7.2 Hiding built-in views

The JPPF administration console is made of a number of built-in views and components assembled together. These views and components can be hidden by specifying configuration properties in the form:

```
jppf.admin.console.view.<view_name>.enabled = true | false
```

where *view_name* is the name of a built-in view, among those listed below:

```
# the server chooser combobox at the top of the UI
jppf.admin.console.view.ServerChooser.enabled = true
# the status bar at the bottom of the UI
jppf.admin.console.view.StatusBar.enabled = true
# the main tabbed pane containing all admin and monitoring views
jppf.admin.console.view.Main.enabled = true
# the tabbed pane containing all topology views
jppf.admin.console.view.Topology.enabled = true
# the topology tree view
jppf.admin.console.view.TopologyTree.enabled = true
# the topology graph view
jppf.admin.console.view.TopologyGraph.enabled = true
# the topology JVM health view
jppf.admin.console.view.TopologyHealth.enabled = true
# the job data view
jppf.admin.console.view.JobData.enabled = true
# the server statistics view
jppf.admin.console.view.ServerStats.enabled = true
# the charts views (tabbed pane)
jppf.admin.console.view.Charts.enabled = false
# the charts configuration view
jppf.admin.console.view.ChartsConfig.enabled = true
# the server load-balancing configuration view
jppf.admin.console.view.LoadBalancing.enabled = true
```

Note that, when left unspecified, any of these properties will default to true (enabled).

7.7.3 Pluggable views

7.7.3.1 Implementing a custom view

It is now possible to add user-defined, pluggable views to the JPPF administration and monitoring console. A pluggable view is a class which extends the abstract class [PluggableView](#) and overrides its `getUIComponent()` method. [PluggableView](#) is defined as follows:

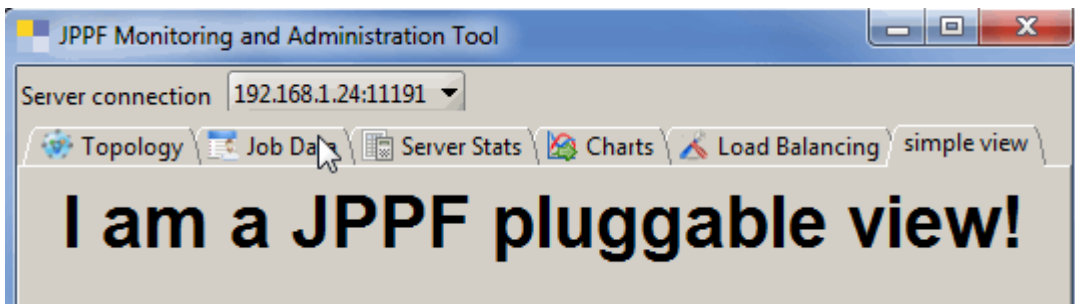
```
public abstract class PluggableView {
    // Get the GUI component which contains the view
    public abstract JComponent getUIComponent();
    // Get the TopologyManager associated with the administration console
    public final TopologyManager getTopologyManager()
}
```

The `getUIComponent()` method returns a [JComponent](#), which will be added as a tab to one of the tabbed panes of the admin console. It can be any subclass of `JComponent`.

As an example, here is a very simple implementation that displays a single label:

```
public class SimpleView extends PluggableView {
    @Override public JComponent getUIComponent() {
        JLabel label = new JLabel("I am a JPPF pluggable view!");
        label.setFont(new Font("Arial", Font.BOLD, 36));
        JPanel panel = new JPanel();
        panel.add(label, BorderLayout.NORTH);
        return panel;
    }
}
```

When added to the console's main tabbed pane, it will look like this:



The `getTopologyManager()` method of [PluggableView](#) allows you to have access to the JPPF grid topology, and register one or more listeners to receive notifications of changes in the topology, as seen in **Development guide > Grid topology monitoring**. For example, expanding from the code sample above:

```
public class SimpleView extends PluggableView {
    @Override public JComponent getUIComponent() {
        JPanel panel = new JPanel();
        JLabel label = new JLabel("I am a JPPF pluggable view!");
        label.setFont(new Font("Arial", Font.BOLD, 36));
        panel.add(label, BorderLayout.NORTH);
        getTopologyManager().addTopologyListener(new TopologyListenerAdapter() {
            @Override public void driverAdded(TopologyEvent e) {
                System.out.println("added driver " + e.getDriver().getDisplayName());
            }
            @Override public void driverRemoved(TopologyEvent e) {
                System.out.println("removed driver " + e.getDriver().getDisplayName());
            }
        });
        return panel;
    }
}
```

Please note that you will need the `jppf-admin.jar` library in your build path / class path at compile time and runtime.

7.7.3.2 Console integration

Integrating a pluggable view into the administration console is done with a number of configuration properties of the form:

```
jppf.admin.console.view.<view_name>.<attribute> = <value>
```

Where view_name is an arbitrary name given to the pluggable view. The possible attributes and their values are defined as follows:

```
# enable / disable the custom view. defaults to true (enabled)
jppf.admin.console.view.SimpleView.enabled = true
# name of a class extending org.jppf.ui.plugin.PluggableView
jppf.admin.console.view.SimpleView.class = org.jppf.example.pluggableview.SimpleView
# the title for the view
jppf.admin.console.view.SimpleView.title = simple view
# path to the icon for the view
#jppf.admin.console.view.SimpleView.icon =
# the tabbed pane the view is attached to
jppf.admin.console.view.SimpleView.addto = Main
# the position at which the custom view is inserted withing the enclosing tabbed pane
# a negative value means insert at the end; defaults to -1 (insert at the end)
jppf.admin.console.view.SimpleView.position = -1
# whether to automatically select the view; defaults to false
jppf.admin.console.view.SimpleView.autoselect = true
```

Since the view can only be attached to a tabbed pane, the possible values for the "addto" attribute are:

- *Main*: the main tabbed pane which contains all administration and monitoring views
- *Topology*: the tabbed pane containing all topology views
- *Charts*: the tabbed pane which contains the user-defined charts and the charts configuration view

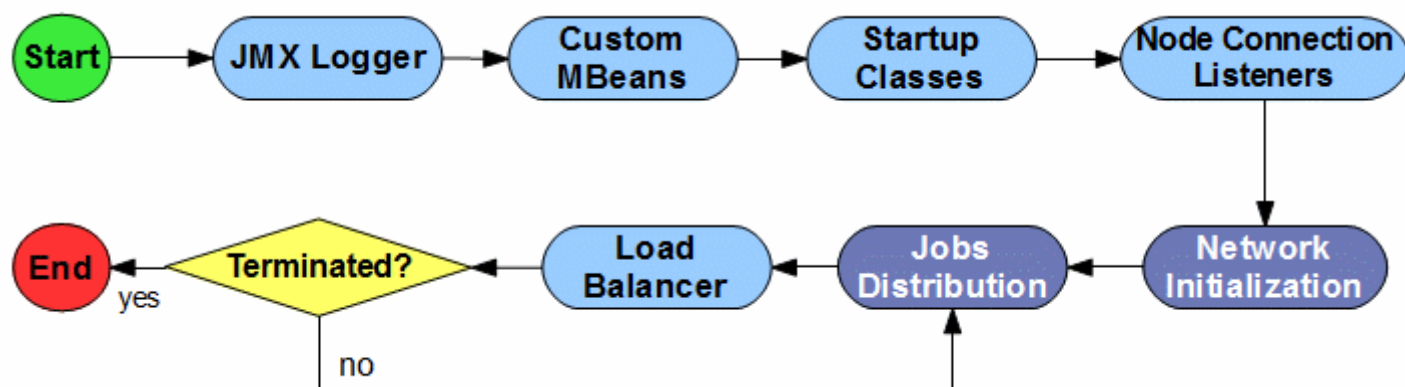
7.7.3.3 Related sample

The [events log view demo](#) provides a complete and nice looking example of a pluggable view.

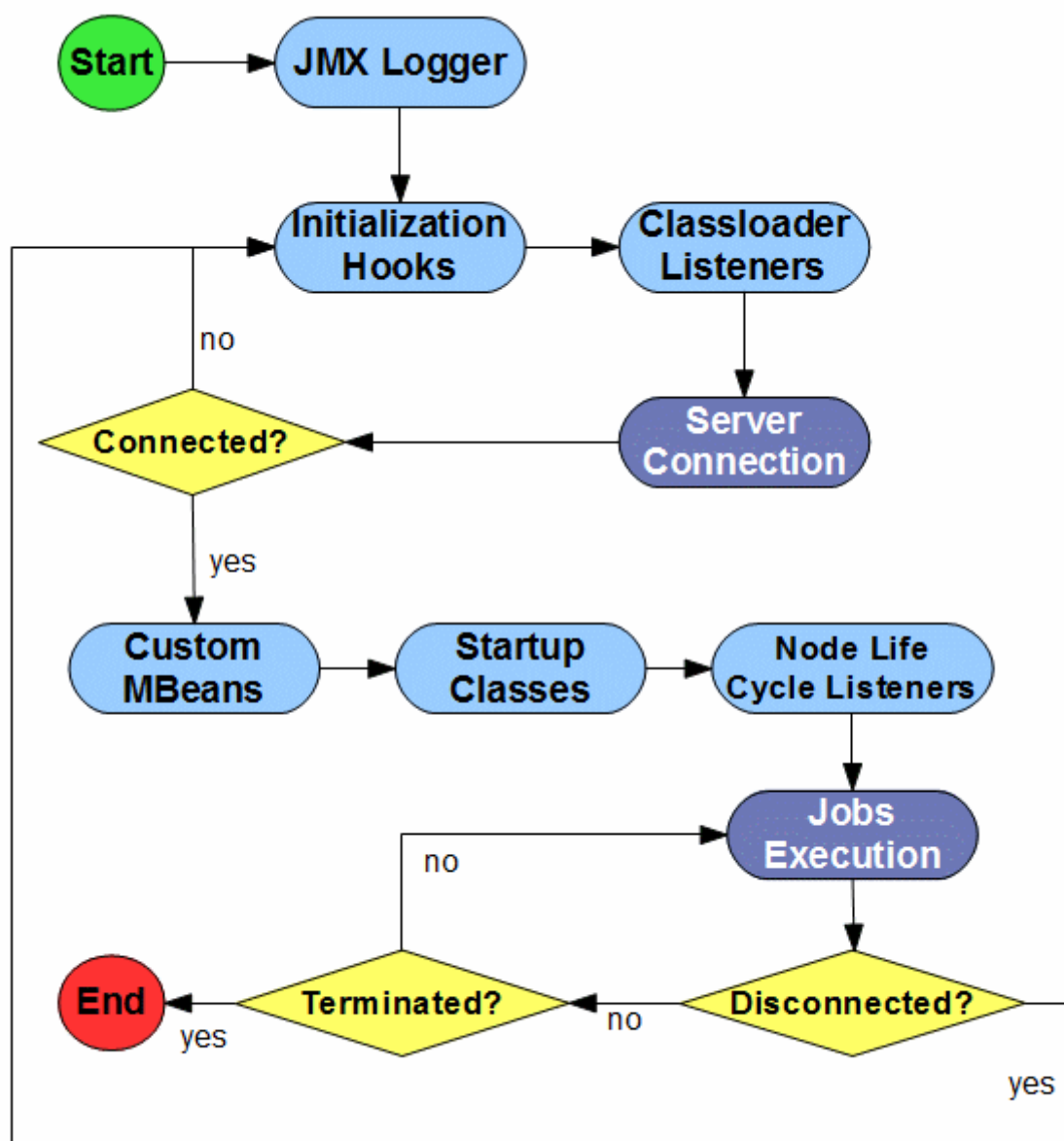
7.8 Flow of customizations in JPPF

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.

7.8.1.1 JPPF driver



7.8.1.2 JPPF node



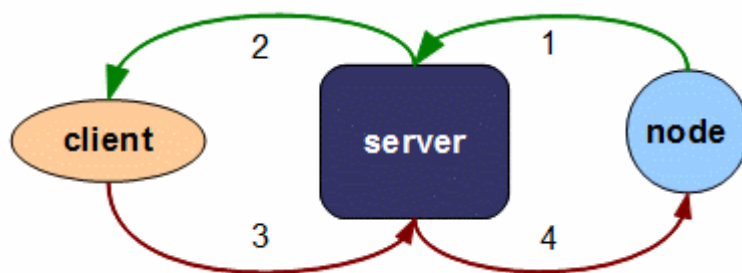
8 Class Loading In JPPF

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 identified 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.

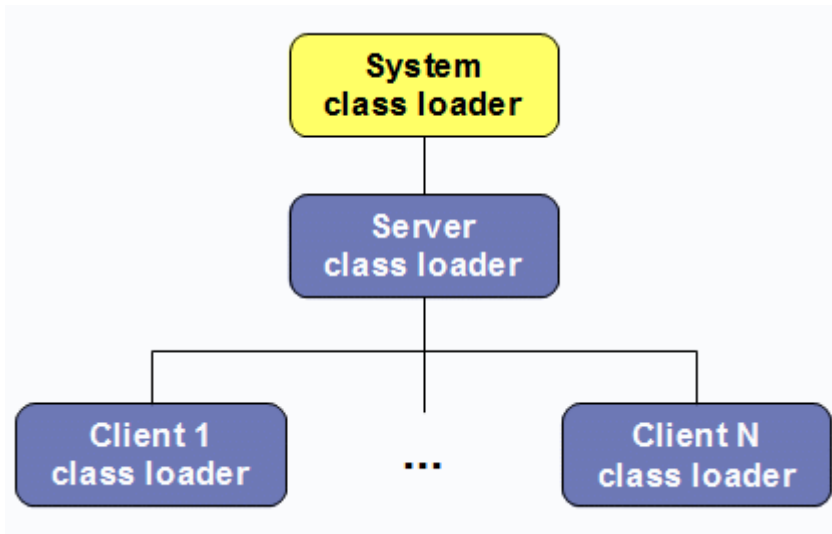
This use case is a simplification of the overall class loading mechanism in JPPF, however it illustrates what actually takes place when a task is executed in a node. This also raises a number of questions that need clarification:

- 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.

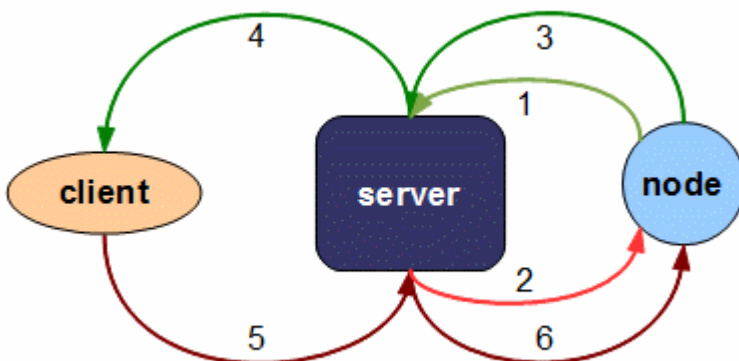
The server class loader is a concrete implementation of the class [AbstractJPPFClassLoader](#) and provides remote access to classes and resources in the server's classpath. It is created at the time the node establishes a connection to the server. It is also discarded when the node disconnects from 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:

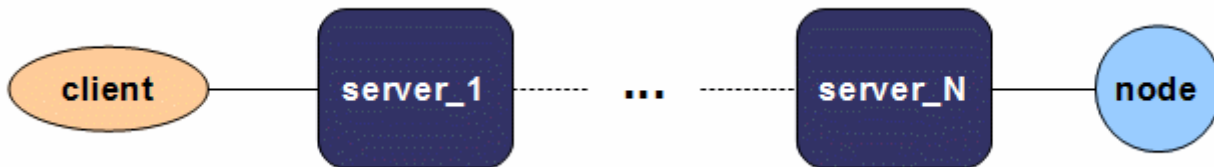


Here, the first two steps are initiated by the server class loader, as a result of the client class loader delegating to its parent. What is missing from this picture are the calls to the system class loader, since they are only meaningful if the requested class is in the node's local classpath.

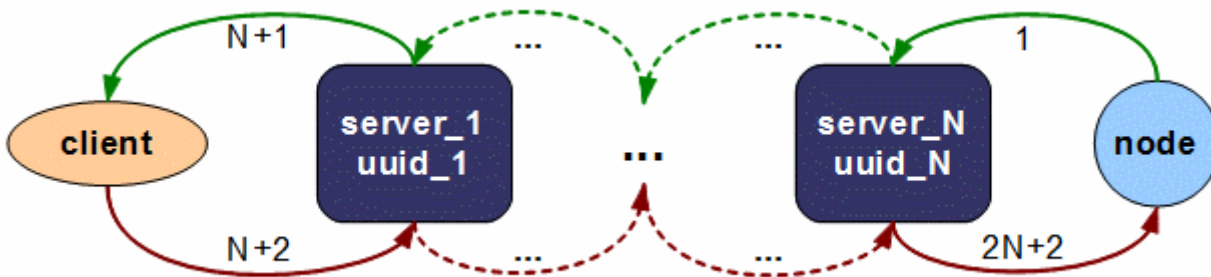
8.3 Relationship between UUIDs and class loaders

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 executed 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 addition 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 resources cache can be enabled or disabled with a configuration property:

```
# whether the cache is enabled, defaults to 'true'
jppf.resource.cache.enabled = true
```

The type of storage for these resources can also be configured:

```
# 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 overridden 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 loading 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 the 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, multiple 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 forwarded 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 delegation 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 specify 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 facility (such as having an FTP server on the JPPF server or 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

Keep in mind that, when `resetTaskClassLoader()` is invoked, the old task class loader is invalidated (closed) and should no longer be used. The class loader instance returned by the method must be used instead. Here is an example usage in a `NodeLifecycleListener`:

```
public class MyNodeListener extends NodeLifecycleListenerAdapter {
    @Override public void jobHeaderLoaded(NodeLifecycleEvent event) {
        try {
            URL url = ...;
            // the old class loader is invalidated (closed) and a new one is created
            AbstractJPPFClassLoader newCL =
                (AbstractJPPFClassLoader ) event.getNode().resetTaskClassLoader();
            newCL.addURL(url);
            URL[] urls = newCL.getURLs();
            // display the added urls
            System.out.println("list of urls: " + Arrays.asList(urls));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

8.7 Related sample

Please look at the [Extended Class Loading](#) sample in the JPPF samples pack.

9 Load Balancing

Load-balancing in JPPF relates to the way jobs are split into sub-jobs, and how these sub-jobs are dispatched to the nodes by a driver, or to the drivers by a client, for execution in parallel. Each sub-job contains a distinct subset of the tasks in the original job. Sub-jobs are also referred to as task bundles or job dispatches, all these terms are equivalent in terms of distribution of jobs into a JPPF grid topology.

9.1 What does it do?

9.1.1 Definition

In JPPF, the goal of a load-balancer is to compute the distribution of the tasks in a job, to one or more drivers on the client side, or to one or more nodes on the server side, in order to optimize the performance of the job's execution.

Practically, a load-balancer will determine how the tasks in a job are split into multiple disjoint subsets, where each subset is sent to a separate driver or node. Since the tasks in a job are known and ordered, it is enough for the load-balancer to compute the size of the subsets.

In JPPF terminology, the tasks subsets are called *task bundles* or just *bundles*, and the code that computes the bundle size is called a *load-balancing algorithm* or just *algorithm* or *bundler* from there on. JPPF provides a number of built-in algorithms, along with an API that allows you to define your own algorithms.

9.1.2 Impact on grid resources usage

Beyond splitting jobs into tasks bundles to send to the nodes, the load-balancer has a significant impact on how the resources in the grid infrastructure will be used:

- shape of the network traffic: sending the tasks one by one or in larger bundles will directly influence the number and frequency of data packets sent over the network
- CPU utilization: if the number of tasks sent to a node is less than its number of processing threads, then the CPU may be under-utilized
- tasks wait time: contrary to CPU under-utilization, when more tasks than the number of processing threads are sent to a node, then some of the tasks may be waiting for a thread to become available, when they could have been sent to another node instead
- heap/memory usage: the more tasks are sent at once, the more memory they will consume. Limiting the bundle size can help avoid out-of-memory conditions.

9.1.3 Server-side vs client-side load balancing

Load-balancing is performed in both the JPPF clients and servers. The question which arises is then: *what do we balance against?* JPPF introduces the generic notion of *execution channel*, or just *channel*, which has a different meaning depending on where it is applied.

For a server, an execution channel can be a connection to a node, or a connection to another server. For example, if server A has 3 nodes connected, server B has 2 nodes, servers A and B are connected to each other, then server A will load-balance against 4 channels (3 nodes + server B) and server B will against 3 channels (2 nodes + server A).

For a client, an execution channel is either a connection to a server or a local executor, knowing that a client can have any combination of one or more connections to a single server, one or more connections to multiple servers or a single local executor. For example, if a client has 3 connections to server A, 2 connections to server B and its local executor enabled, then it will load-balance against 6 execution channels.

Adding to this, let's remember that both clients and servers can handle multiple jobs concurrently, and that these jobs can vary vastly in how they use grid resources, and we can see that load-balancing is a fairly non-trivial task.

9.1.4 Qualitative characteristics of the algorithms

Load-balancing algorithms can compute a bundle size in many different ways, with various levels of complexity and access to multiple sources of information their computations can be based on. To qualify these algorithms for a better understanding of how they work, JPPF uses three distinguishing characteristics:

Static vs. adaptive: a static algorithm always returns the same bundle size for a given (node, job) pair, whereas an adaptive algorithm will adjust the bundle size based on feedback from, and information on, the node and/or job.

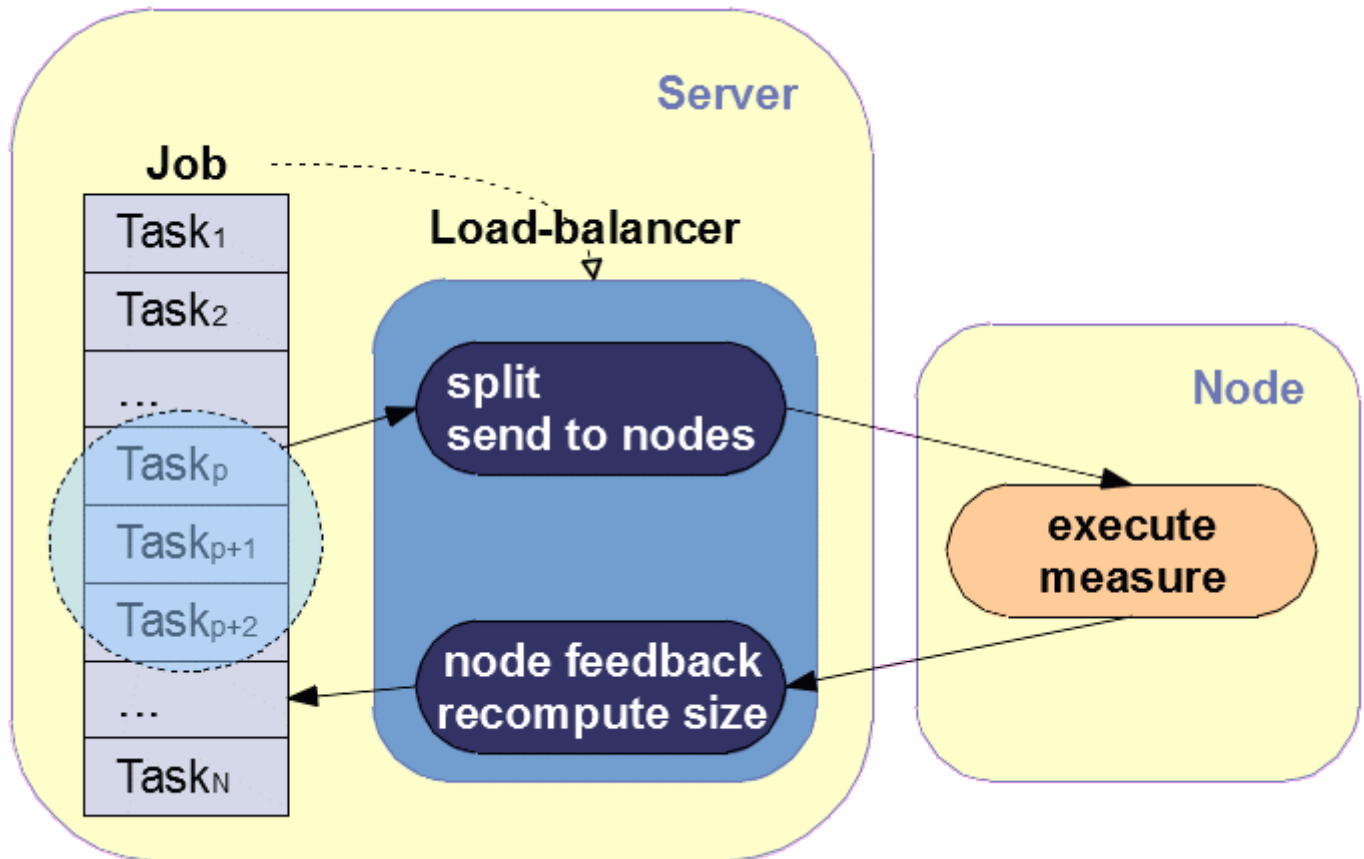
Deterministic vs. heuristic: a deterministic algorithm does not use any random step in its computations. In other words, given the same input and information, it will always return the same value. A heuristic algorithm, on the other hand, will make informed random guesses while exploring the solutions space.

Local vs. global: a local algorithm only computes the bundle size for a single channel, based on information that only applies to this channel (hence the locality), whereas a global algorithm will recompute or at least impact the bundle size computation on all the available channels.

9.2 Load balancing API

9.2.1 How it works

The basic flow of a load-balancer is shown in the following figure:



As we can see, from a high-level perspective it is made of a continuous feedback loop:

- the load-balancer determines which tasks of a job to send to a node, according to its last computed bundle size
- upon receiving the tasks results from the node, it also receives information on the execution performance: number of tasks, total round-trip time, and optionally the total accumulated elapsed time for tasks execution, along with the network transport overhead
- information on the job being split can optionally be injected, allowing the load-balancer to recompute the bundle size based of the job's state and properties.

9.2.2 Bundler

In JPPF, all load-balancing algorithms implement the [Bundler](#) interface, defined as follows:

```
public interface Bundler<T extends LoadBalancingProfile> {
    // Get the last computed bundle size
    int getBundleSize();
    // Provide feedback from a channel after execution of a set of tasks
    void feedback(final int nbTasks, final double totalTime);
    // Get the timestamp at which this bundler was created
    long getTimestamp();
    // Perform context-independent initializations
    void setup();
    // Release the resources used by this bundler
    void dispose();
    // Get the parameters profile used by this load-balancer
    T getProfile();
}
```

For the `feedback(int, double)` method, the feedback data consists in a number of tasks that were executed, along with their total execution time in milliseconds. The execution time includes the network round trip between node and server or

between server and client, along with the serialization and deserialization time, and any other overhead time from JPPF-specific processing.

The `getTimestamp()` methods provides the bundler's creation timestamp. It is used for the purpose of dynamically updating the load-balancing settings, as can be done with [JMX API calls](#) or from the [administration console](#). The server or client will compare this timestamp with the last modification date of the settings, and instantiate a new bundler based on these settings whenever the settings are newer.

The `setup()` and `dispose()` methods are lifecycle methods and are called right after a bundler is created, and just before it is dismissed, respectively. Their purpose is to allow the bundler to configure/load any resources it will use during its life span, and clean these resources up when it is terminated.

Note: it is important to remember that JPPF creates a [Bundler](#) instance for each channel it load-balances against. The server will associate a distinct bundler with each node connection. Similarly, the client will create a distinct bundler for each server connection, plus another for its local executor.

For all practical purposes, it is simpler and easier to extend the [AbstractBundler](#) class, rather than to implement [Bundler](#) directly. [AbstractBundler](#) is defined as follows:

```
public abstract class AbstractBundler<T extends LoadBalancingProfile>
    implements Bundler<T> {

    // Creates a new instance with the specified parameters profile
    public AbstractBundler(T profile)
    // Get the max bundle size that can be used for this bundler
    public int maxSize()
    // This implementation does nothing and should be overridden in subclasses
    // that compute the bundle size based on the feedback from the nodes
    @Override public void feedback(int bundleSize, double totalTime)
    // Get the timestamp at which this bundler was created
    @Override public long getTimestamp()
    // Perform context-independent initializations
    @Override public void setup()
    // Release the resources used by this bundler
    @Override public void dispose()
    // Get the parameters of the algorithm
    @Override public T getProfile()

}
```

In addition to providing default implementations for a number of methods in `Bundler`, it also adds a `maxSize()` method, which provides a hint as to the maximum bundle size the bundler should return. This addresses the problem that, if too many tasks are sent to a single channel, there will be no fair distribution of the tasks and the overall grid performance will suffer from it. As mentioned, this is just a hint and there is no obligation to use it.

9.2.3 BundlerEx

The [BundlerEx](#) interface extends [Bundler](#) and provides an additional `feedback()` method two with more parameters:

```
public interface BundlerEx<T extends LoadBalancingProfile> extends Bundler<T> {

    // Feedback the bundler with the performance result of a task bundle
    void feedback(int nbTasks, double totalTime, double accumulatedElapsed,
        double overheadTime);

}
```

As for `Bundler.feedback(int, double)`, the `nbTasks` and `totalTime` parameters represent the number of tasks in the bundle and the total round-trip time of the bundle, respectively. The purpose of the other two parameters is to provide greater accuracy when computing the performance of the task bundle's execution:

accumulatedElapsed is the sum of the execution elapsed time of all the tasks in the bundle. It differs from the total execution time in that it considers the execution *as if it occurred on a single thread*.

overheadTime measures the sum of the network transport time and the JPPF overhead time, which also includes serialization and deserialization. The total time spent actually executing the tasks can then be calculated as:

$$\text{executionTime} = \text{totalTime} - \text{overheadTime}$$

To illustrate these parameters, let's consider a node that has 2 threads. For clarity's sake, let's imagine the tasks all have the same duration `d`. We distinguish several scenarios:

1) We send 1 task to the node: both the total execution time and the accumulated elapsed time will be equal to d:

```
executionTime = d
accumulatedElapsed = d
```

2) We send 2 tasks to the node: since the node has 2 threads, the 2 tasks will execute concurrently, and we will have:

```
executionTime = d
accumulatedElapsed = 2*d
```

3) We send 3 tasks to the node: the first two tasks will execute concurrently, while the third task will wait until a thread becomes available, i.e. until one of the first two tasks completes. We will then have:

```
executionTime = 2*d
accumulatedElapsed = 3*d
```

If the algorithm is aware of the number of threads in the node, then it can easily, in its computations, get rid of the disturbance introduced by threads remaining idle by lack of tasks to execute.

Note: at runtime, when JPPF detects that a bundler implements `BundlerEx`, it will call the `BundlerEx.feedback()` method instead of the `Bundler.feedback()` method.

The abstract class [AbstractAdaptiveBundler](#) implements [BundlerEx](#), and for all practical purposes it will be easier to extend it than to implement [BundlerEx](#) directly:

```
public abstract class AbstractAdaptiveBundler<T extends LoadBalancingProfile>
    extends AbstractBundler<T> implements BundlerEx<T>, NodeAwareness, JobAwarenessEx {
    // the last computed bundle size
    protected int bundleSize;

    // Creates a new instance with the specified parameters profile
    public AbstractAdaptiveBundler(T profile)

    // receive feedback from a node
    @Override public void feedback(int size, double totalTime,
                                   double accumulatedElapsed, double overheadTime)

    // get the last computed bundle size
    @Override public int getBundleSize()

    @Override public JPPFSystemInformation getNodeConfiguration()

    @Override public void setNodeConfiguration(JPPFSystemInformation nodeConfiguration)

    @Override public JPPFDistributedJob getJob()

    @Override public void setJob(JPPFDistributedJob job)

    @Override public void dispose()

    @Override public int maxSize()
}
```

[AbstractAdaptiveBundler](#) has a default implementation of `BundlerEx.feedback()` which computes a synthetic `totalTime` where the idle threads overhead is removed, then delegates to `Bundler.feedback()` with this synthetic value.

9.2.4 Parameters profile

A load-balancing algorithm may use zero or more parameters, which can be specified in the configuration of a JPPF driver or client. These parameters are encapsulated in an implementation of the [LoadBalancingProfile](#) interface, defined as follows:

```
public interface LoadBalancingProfile extends Serializable {  
}
```

As we can see, there is no method in this interface, which makes [LoadBalancingProfile](#) a marker interface for all purposes and intents. As a convenience, an abstract implementation is provided: the [AbstractLoadBalancingProfile](#) class.

The link between a bundler and its parameters profile is provided by the `Bundler.getLoadBalancingProfile()` method. It is also generally convenient to have the [LoadBalancingProfile](#) passed in the bundler's constructor, as is the case for [AbstractBundler](#) and [AbstractAdaptiveBundler](#).

9.2.5 Bundler provider

JPPF relies on the Service Provider Interface (SPI) mechanism to discover the defined load-balancing algorithm. To this effect, it is required that, for each algorithm, an implementation of the [JPPFBundlerProvider](#) interface be given:

```
public interface JPPFBundlerProvider<T> extends LoadBalancingProfile {  
    // Get the name of the algorithm defined by this provider  
    String getAlgorithmName();  
  
    // Create a bundler instance using the specified parameters profile  
    Bundler<T> createBundler(T profile);  
  
    // Create a bundler profile containing the parameters of the algorithm  
    T createProfile(TypedProperties configuration);  
}
```

Notes:

a) the method `getAlgorithmName()` must return a name that is unique accross all algorithms, otherwise only the last discovered algorithm will be kept.

b) the parameters profile provided in the `createBundler()` method is created by invoking the method `createProfile(TypedProperties)`.

c) the `TypedProperties` object provided in the `createProfile()` method contains only the configuration properties for a given parameters profile, where the names of the properties are stripped of their JPPF-specific prefix. For instance, if the configuration contains:

```
jppf.load.balancing.profile = myProfile1  
jppf.load.balancing.profile.myProfile1.param1 = value1  
jppf.load.balancing.profile.myProfile2.param2 = value2
```

then the configuration provided to `createProfile()` will contain a single property definition without prefix:

```
param1 = value1
```

d) When an algorithm doesn't use any parameter, then you can implement an associated bundler provider using [LoadBalancingProfile](#) as the generic profile type, and returning `null` in the `createProfile()` method:

```
public class MyProvider implements JPPFBundlerProvider<LoadBalancingProfile> {  
    @Override public String getAlgorithmName() { return "myAlgorithm"; }  
  
    @Override public Bundler createBundler(LoadBalancingProfile profile) {  
        return new MyAlgorithm(); // profile is not used  
    }  
  
    @Override public LoadBalancingProfile createProfile(TypedProperties configuration) {  
        return null; // configuration is not used  
    }  
}
```

Finally, to enable the discovery of the algorithm, it is required to create, in the META-INF/services folder, a file named: `org.jppf.load.balancing.spi.JPPFBundlerProvider`

In this file, enter the fully qualified name of the bundler provider implementation, for each existing algorithm, on a separate line, as in this example:

```
# my custom load-balancer algorithm
com.example.MyProvider
# the JPPF "manual" algorithm
org.jppf.load.balancer.spi.FixedSizeBundlerProvider
```

9.2.6 Channel awareness

A load-balancer that wishes to receive information about its associated channel, to base its computations on, should implement the [ChannelAwareness](#) interface:

```
public interface ChannelAwareness {
    // Get the corresponding node's system information
    JPPFSystemInformation getChannelConfiguration();

    // Set the corresponding node's system information
    void setChannelConfiguration(JPPFSystemInformation channelConfiguration);
}
```

As we can see, this interface allows JPPF to set an attribute of type [JPPFSystemInformation](#) onto the load-balancer. The properties contained in this attribute are described in full details in the [Execution policy properties](#) section.

The `setChannelConfiguration()` method is a bundler life cycle callback, invoked internally by the JPPF client or server, when the channel initially establishes a connection (handshake) or when one or more properties in its configuration, including the number of threads, are changed dynamically.

Notes:

- when a bundler implements the [ChannelAwareness](#) interface, a JPPF client or server will automatically recognize it and call its `setChannelConfiguration()` method as appropriate.
- the [AbstractAdaptiveBundler](#) class already implements the [ChannelAwareness](#) interface, and its implementation of `setChannelConfiguration()` sets a [channelConfiguration](#) attribute with the provided [JPPFSystemInformation](#) and also extracts the number of threads as applicable into a [nbThreads](#) attribute of type `int`.

9.2.7 Job awareness

A bundler can also receive information on the job being distributed over the server node channels by implementing the [JobAwareness](#) interface:

```
public interface JobAwarenessEx {
    // Get the current job for which load-balancing is being performed
    JPPFDistributedJob getJob();

    // Set the current job for which load-balancing is being performed
    void setJob(JPPFDistributedJob job);
}
```

This interface allows the client or server to set an attribute of type [JPPFDistributedJob](#) onto the bundler.

The `setJob()` method is a bundler life cycle callback, invoked internally by the JPPF client or server, whenever a set of tasks from a job is dispatched to a channel.

As for [ChannelAwareness](#), a bundler that implements [JobAwareness](#) will be automatically recognized and its `setJob()` method invoked at the appropriate times.

[AbstractAdaptiveBundler](#) also implements [JobAwareness](#) and its implementation of `setJob()` sets a [job](#) attribute of type [JPPFDistributedJob](#).

9.2.8 Simple code example: the "manual" algorithm

To try and translate these concepts into something more concrete, we will now walk through the implementation of the simplest of the JPPF built-in algorithms: the "manual" algorithm. The "manual" algorithm is a [static, global and deterministic](#) algorithm which always returns the same fixed bundle size for all the channels. It uses a single parameter named "size" and is configured as follows in a JPPF configuration file:

```
# name of the load-balancing algorithm
jppf.load.balancing.algorithm = manual
# name of the set of parameter (profile) for the algorithm
jppf.load.balancing.profile = manual_profile
# "manual_profile" profile
jppf.load.balancing.profile.manual_profile.size = 20
```

Remember, during the discovery and creation of the load-balancers, this configuration will be stripped down to only the essential information needed:

```
size = 20
```

Based on this, we implement a the profile class [FixedSizeProfile](#) as follows:

```
public class FixedSizeProfile extends AbstractLoadBalancingProfile {
    private final int size;

    // Initialize this profile with values read from the specified configuration
    public FixedSizeProfile(final TypedProperties config) {
        int n = config.getInt("size", 1);
        this.size = (n < 1) ? 1 : n;
    }

    // Get the bundle size
    public int getSize() { return size; }
}
```

As we can see, this profile simply extracts the "size" parameter from the configuration at construction time and exposes it to other classes with a getter.

Given its simplicity, the algorithm does not need job awareness nor channel awareness, therefore it can be implemented by extending [AbstractBundler](#):

```
public class FixedSizeBundler extends AbstractBundler<FixedSizeProfile> {
    // Initialize this bundler
    public FixedSizeBundler(FixedSizeProfile profile) {
        super(profile);
    }

    // Returns the bundle size statically assigned in the configuration
    @Override
    public int getBundleSize() {
        return profile.getSize();
    }
}
```

Note here that we leave the implementation of the `feedback(int, double)` method to the superclass, which does nothing (empty implementation).

The associated [JPPFBundlerProvider](#), the [FixedSizeBundlerProvider](#) class, is then implemented like this:

```
public class FixedSizeBundlerProvider implements JPPFBundlerProvider<FixedSizeProfile> {
    // Create a bundler instance using the specified parameters profile
    @Override
    public Bundler createBundler(FixedSizeProfile profile) {
        return new FixedSizeBundler(profile);
    }

    // Create a bundler profile containing the parameters of the algorithm
    @Override
    public FixedSizeProfile createProfile(TypedProperties configuration) {
        return new FixedSizeProfile(configuration);
    }

    // Get the name of the algorithm defined by this provider
    @Override
    public String getAlgorithmName() {
        return "manual";
    }
}
```

Finally, for JPPF to discover the algorithm, we add the bundler provider's fully qualified class name to the service file META-INF/services/org.jppf.load.balancing.spi.JPPFBundlerProvider:

```
# the "manual" algorithm
org.jppf.load.balancer.spi.FixedSizeBundlerProvider
```

9.3 Built-in algorithms

9.3.1 "manual"

As seen in the [code example section](#), the "manual" algorithm is a static, global and deterministic algorithm which always uses the same bundle size for all the channels. It is equivalent to a round-robin mechanism.

Here is an example configuration:

```
# name of the load-balancing algorithm
jppf.load.balancing.algorithm = manual
# name of the set of parameter (profile) for the algorithm
jppf.load.balancing.profile = fixed_size
# "manual_profile" profile
jppf.load.balancing.profile.fixed_size.size = 20
```

9.3.2 "nodethreads"

The "nodethreads" algorithm is an adaptive, local and deterministic algorithm which computes the bundle size based on a channel's number of processing threads. It uses a single parameter named "multiplicator" such that:

```
bundle_size = multiplicator * processing_threads
```

Therefore, the bundle size is always a multiple of the channel's number of processing threads.

Due to its computations being based of the number of processing threads, this algorithm can only effectively apply to a node, because server channels do not have a notion of processing threads. Therefore, it should only be used on the server-side. If used on the client-side, JPPF will "force" a number of threads equal to 1 for all channels, making the algorithm equivalent to a "manual" algorithm.

This algorithm is adaptive because the number of processing threads in a node can be update dynamically and the updates will be reported to the load-balancer on the server side. This is also why this algorithm implements the [ChannelAwareness](#) interface.

Example configuration:

```
# name of the load-balancing algorithm
jppf.load.balancing.algorithm = nodethreads
# name of the set of parameter (profile) for the algorithm
jppf.load.balancing.profile = threads
# "manual_profile" profile
jppf.load.balancing.profile.threads.multiplicator = 1
```

9.3.3 "autotuned"

This algorithm is heuristic in the sense that it determines a good solution for the number of tasks to send to each channel, while the solution may not be the optimal one. It is loosely based on a simulated annealing algorithm. The heuristic part is provided by the fact that this algorithm performs a random walk within the solutions space.

It is a purely adaptive algorithm, based on the known past performance of each channel. It does not rely or know about the characteristics of the channels (i.e. hardware and software configuration). This algorithm is local to each channel, meaning that a separate task bundle size is determined for each channel, independently of the other channels. In fact, the other channels will implicitly adjust due to the performance changes generated each time the bundle size changes.

It starts using the bundle size defined in property file and changes it to find a better performance. The algorithm waits for some execution results to get a mean execution time, and then makes a change to the bundle size. Each time a change is done, it is done over a smaller range randomly.

This algorithm uses the following parameters:

- **"size"**: this is the initial bundle size to start with, to bootstrap the algorithm.
- **"minSamplesToAnalyse"**: the minimum number of samples that must be collected before an analysis is triggered.
- **"minSamplesToCheckConvergence"**: the minimum number of samples to be collected before checking if the performance profile has changed.
- **"maxDeviation"**: the percentage of deviation of the current mean to the mean when the system was considered stable.
- **"maxGuessToStable"**: the maximum number of guesses of number generated that were already tested for the algorithm to consider the current best solution stable.
- **"sizeRatioDeviation"**: this parameter defines the multiplicity used to define the range available to random generator, as the maximum.

- **"decreaseRatio"**: this parameter defines how fast it will stop generating random numbers. This is essential to define the size of the universe that is explored. Greater numbers make the algorithm stop sooner. Just as example, if the best solution is between 0-100, the following might occur (unless maxGuessToStable is small):
 - 1 => 2 max guesses
 - 2 => 5 max guesses
 - 0.5 => 9 max guesses
 - 0.1 => 46 max guesses
 - 0.05 => 96 max guesses

Example configuration:

```
# name of the load-balancing algorithm
jppf.load.balancing.algorithm = autotuned
# name of the set of parameter (profile) for the algorithm
jppf.load.balancing.profile = autotuned
# parameters
jppf.load.balancing.profile.autotuned.size = 5
jppf.load.balancing.profile.autotuned.minSamplesToAnalyse = 100
jppf.load.balancing.profile.autotuned.minSamplesToCheckConvergence = 50
jppf.load.balancing.profile.autotuned.maxDeviation = 0.2
jppf.load.balancing.profile.autotuned.maxGuessToStable = 50
jppf.load.balancing.profile.autotuned.sizeRatioDeviation = 1.5
jppf.load.balancing.profile.autotuned.decreaseRatio = 0.2
```

9.3.4 "proportional"

The "proportional" algorithm is an adaptive, global and deterministic load-balancing algorithm. As for the "autotuned" algorithm, it is purely adaptive and based solely on the known past performance of the channels.

The computation is performed without a random part. Each bundle size is determined in proportion to the mean task execution time to the power of N. Here, N is one of the algorithm's parameters, called "proportionality factor". The mean time is computed as a moving average over the last M executed tasks for a given channel. M is another algorithm parameter, called "performance cache size".

Also, and contrary to the "autotuned" approach, this algorithm is global: the bundle size for each channel depends on that of the other channels.

Example configuration with the default values:

```
# name of the load-balancing algorithm
jppf.load.balancing.algorithm = proportional
# name of the set of parameter (profile) for the algorithm
jppf.load.balancing.profile = prop

# algorithm parameters
jppf.load.balancing.profile.prop.performanceCacheSize = 2000
jppf.load.balancing.profile.prop.proportionalityFactor = 1

# bootstrap parameters
jppf.load.balancing.profile.prop.initialSize = 1e9
jppf.load.balancing.profile.prop.initialMeanTime = 5
```

Note the "initialSize" and "initialMeanTime" parameters, which are used to bootstrap the algorithm the first time a bundler is used.

Description of this algorithm

First, let's define the following variables:

- **n** = current number of channels
- **max** = maximum number of tasks in a job in the current queue state
- **mean_i** = mean execution time for channel i
- **s_i** = number of tasks to send to channel i
- **p** = proportionality factor parameter

We then define:

$$S = \sum_{i=1}^n \left(\frac{1}{\text{mean}_i} \right)^p$$

s_i is then computed as:

$$s_i = \text{max} \times \frac{\left(\frac{1}{\text{mean}_i} \right)^p}{S}$$

Here, we can see that the bundle size for each node is proportional to its contribution to sum S, hence the name of the algorithm. Every time performance data is fed back to the load-balancer, all channel bundle sizes are re-computed.

Note: a noteworthy consequence of the algorithm's implementation is that it will always attempt to dispatch all the tasks in a job at once among the available channels.

9.3.5 "rl2"

The "rl2" algorithm is a deterministic, local and heuristic algorithm which randomly explores the space of bundle sizes and builds a set of possible states of the system it represents over time, in order to determine the states and actions that provide a performance that is as close to optimal as possible.

It uses the following parameters:

- **"performanceCacheSize"**: the maximum size of the performance samples cache. The lower, the more sensitive the bundler is to changes in the tasks performance profile, i.e. it adapts faster at the potential risk of over-adjusting to non-significant changes
- **"performanceVariationThreshold"**: the minimum variation of the mean execution time that triggers a reset of the states of the system, thereby causing the algorithm to restart its learning phase. It is used to detect when the performance has significantly changed, for example when a new type of jobs is submitted to the grid
- **"minSamples"**: the minimum number of states (bundle size associated with a mean execution time) to collect randomly before switching to the next phase of the algorithm
- **"maxSamples"**: the maximum number of randomly collected states before the random exploration phase ends
- **"maxRelativeSize"**: the maximum value of the bundle size, expressed as a fraction of the size of the current job being evaluated. This avoids sending all the tasks of a job to a single node, which would defeat the purpose of parallelization.

The algorithm performs the following actions, when its `feedback()` method is invoked:

Let a state be a couple (bundleSize, meanTime) where meanTime is the best (i.e. lowest) known mean execution time for a bundle of size bundleSize.

Let nbStates be the number of known states

Let maxSize be `currentJobSize * maxRelativeSize`

1) Detect whether a significant performance change has occurred. This happens for instance when a new kind of jobs is submitted to the grid, where the tasks in these jobs take a significantly longer time to execute. When this happens, the algorithm clears all its known states, to "forget" what it has learned about previous jobs' performance, and restarts its learning cycle. The condition for a significant performance change is expressed as:

$(\text{previousMean} - \text{currentMean}) / \text{previousMean} < \text{performanceVariationThreshold}$

where previousMean and currentMean represent the mean execution time of the performance cache before and after it has been updated with the latest feedback.data

2) When `nbStates < minSamples`, a new bundle size is chosen randomly among the values in the range `[1, maxSize]` for which no state has been collected yet.

3) When `minSamples <= nbStates < maxSamples`, we compute a probability `p` that the next bundle size will be chosen randomly, which decreases linearly with `(maxSamples - nbStates)`. In other terms, the more known states, the less likely it is that the bundle size will be chosen randomly. The probability `p` is computed as:

$p = (\text{maxSamples} - \text{nbStates}) / (1 + (\text{maxSamples} - \text{minSamples}))$

If the bundle size is not chosen randomly, then it is set to the bundle size of the known state with the best performance.

4) When `nbStates >= maxSamples` then the next bundle size is the size of the known state with the best performance.

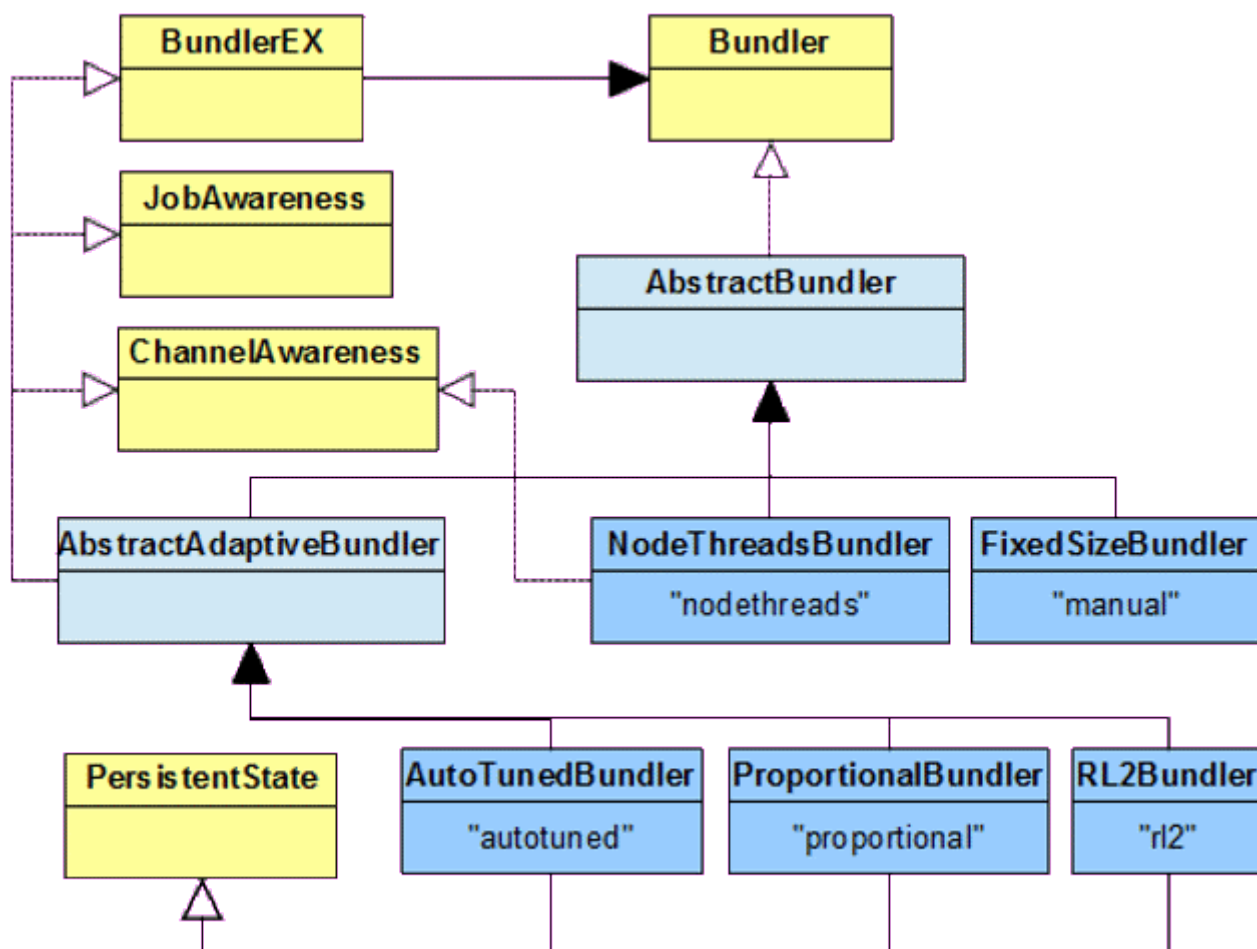
To summarize, we can say that the algorithm goes through 3 phases over time: it first learns the states of the system by randomly walking the space of possible bundle sizes, then continues to learn while increasingly applying its acquired knowledge, and finally considers learning as complete and uses the optimal value it converged to.

Example configuration with default values:

```
# name of the load-balancing algorithm
jppf.load.balancing.algorithm = rl2
# name of the set of parameter (profile) for the algorithm
jppf.load.balancing.profile = rl2_profile
# "rl2" parameters
jppf.load.balancing.profile.rl2_profile.performanceCacheSize = 1000
jppf.load.balancing.profile.rl2_profile.performanceVariationThreshold = 0.75
jppf.load.balancing.profile.rl2_profile.minSamples = 20
jppf.load.balancing.profile.rl2_profile.maxSamples = 100
jppf.load.balancing.profile.rl2_profile.maxRelativeSize = 0.5
```

9.3.6 Class hierarchy

All JPPF's built-in bundlers are implemented in the [org.jppf.load.balancer.impl](#) package. Similarly, the corresponding bundler providers are found in the [org.jppf.load.balancer.spi](#) package.



9.4 Load-balancer state persistence

As seen previously, most of the existing load-balancing algorithms are adaptive and often go through a convergence phase before reaching a state of optimal efficiency. Unfortunately, this state disappears whenever a node, driver or client (depending on where load-balancing applies) is stopped.

As of JPPF 6.0, it is now possible to persist the state of load-balancer instances and restore them whenever a node reconnects to a driver (server-side load-balancing) or when a client reconnects to a driver (client-side load-balancing).

As for many other JPPF features, this state persistence facility is pluggable, which means it is possible to create user-defined implementations. The full details of how to do this are provided in the [Custom load-balancer state persistence](#) section.

9.4.1 Identification of persisted load-balancer states

Persistence and restoration of a load-balancer state requires an identifier that is both unique and repeatable across reconnection and restart of the peers involved (node + driver or driver + client). Using combinations of the components' UUIDs will not work here, since UUIDs only exist as long as each component's JVM is alive and are generated anew at each restart.

Instead, JPPF computes a resilient identifier which includes the IP address of both parties, the related driver's port and other properties found in the JPPF component's configuration. A load-balancer state is further identified by the name of the of its algorithm. For normalization purposes, both parts of the identifier are transformed using a hash function. This function defaults to SHA-1 and can be changed using the following configuration property:

```
jppf.load.balancing.persistence.hash = SHA-1
```

9.4.2 Configuring

A load-balancer state persistence facility is essentially a pluggable implementation of the [LoadBalancerPersistence](#) interface. A driver or JPPFClient instance supports a single persistence implementation at a time, configured via the "jppf.load.balancing.persistence" configuration property:

```
jppf.load.balancing.persistence = <implementation class name> param1 ... paramN
```

where:

- **"implementation class name"** is the fully qualified class name of the [LoadBalancerPersistence](#) implementation
- **"param1 ... paramN"** are optional string parameters used by the persistence implementation

For example, to configure the [built-in file persistence](#) with a root directory named "lb_persistence" in the driver's working directory, we would configure the following:

```
pkg = org.jppf.load.balancer.persistence  
jppf.load.balancing.persistence = ${pkg}.FileLoadBalancerPersistence lb_persistence
```

If no load-balancer persistence is configured, then persistence of load-balancer states is disabled.

9.4.3 Built-in implementations

9.4.3.1 File-driven persistence

File-driven persistence of load-balancer states is implemented in the class [FileLoadBalancerPersistence](#). It relies on a directory structure like this:

```
persistence_root  
|_channel_identifier1  
| |_algorithm1.data  
| |_  
| |_algorithmp1.data  
|_  
|_channel_identifiern  
| |_algorithm1.data  
| |_  
| |_algorithmpn.data
```

Where:

- Each `channel_identifieri` represents a hash of a string concatenated from various properties of the channel. A channel represents a connection between a node and a driver for server-side load-balancing, or between a driver and a client for client-side load-balancing. This id is unique for each channel and resilient over restarts of both related peers, contrary to their uuids, which are recreated each time a component starts. Using a hash also ensures that it can be used as a valid folder name in a file system
- Each `algorithmi` prefix is the hash of the related load-balancing algorithm name. Again, this ensures it can be used to form a valid file name
- Each `algorithmi.data` file represents the serialized state of the related load-balancer

The file-driven persistence is configured as:

```
pkg = org.jppf.load.balancer.persistence
jppf.load.balancing.persistence = ${pkg}.FileLoadBalancerPersistence <root_dir>
```

where `root_dir` can be either an absolute file path, or a path relative to the JVM's current working directory. If it is omitted, it defaults to "**lb_persistence**".

9.4.3.2 Database-driven persistence

Database-driven persistence is implemented in the class [DatabaseLoadBalancerPersistence](#) and stores load-balancer states in a single database table. The table has the following structure:

```
CREATE TABLE <table_name> (
  NODEID varchar(250) NOT NULL,
  ALGORITHMID varchar(250) NOT NULL,
  STATE blob NOT NULL,
  PRIMARY KEY (NODEID, ALGORITHMID)
);
```

Where:

- the **NODEID** column represents a hash of a string concatenated from various properties of the node. This id is unique for each node and resilient over node restarts, contrary to the node uuid, which is recreated each time a node starts
- the **ALGORITHMID** column is a hash of the load-balancer's algorithm name.
- the **STATE** column represents the serialized state of the load-balancer, such as provided by `PersistentState.getState()`

Very important: the table definition should be adjusted depending on the database you are using. For instance, in MySQL the BLOB type has a size limit of 64 KB, thus storing load-balancer states larger than this size will always fail. In this use case, the MEDIUMBLOB or LONGBLOB type should be used instead.

The database-driven persistence is configured as follows:

```
pkg = org.jppf.load.balancer.persistence
jppf.load.balancing.persistence = ${pkg}.DatabaseLoadBalancerPersistence \
  <table_name> <datasource_name>
```

where:

- **table_name** is the name of the table in which to store load-balancer states
- **datasource_name** is the name of a data source defined elsewhere in the configuration. See the [database services](#) section of the documentation for details.

If both are unspecified, they will default to "load_balancer" and "loadBalancerDS", respectively. If only the table name is specified, the data source name defaults to "loadBalancerDS". If the table does not exist, JPPF will attempt to create it. If this fails for any reason, for instance if the user does not have sufficient privileges, then persistence will be disabled.

Here is a full example configuration:

```
# persistence definition
pkg = org.jppf.load.balancer.persistence
jppf.load.balancing.persistence = ${pkg}.DatabaseLoadBalancerPersistence \
    MY_TABLE loadBalancerDS

# datasource definition
jppf.datasource.lb.name = loadBalancerDS
jppf.datasource.lb.driverClassName = com.mysql.jdbc.Driver
jppf.datasource.lb.jdbcUrl = jdbc:mysql://localhost:3306/testjppf
jppf.datasource.lb.username = testjppf
jppf.datasource.lb.password = testjppf
jppf.datasource.lb.minimumIdle = 5
jppf.datasource.lb.maximumPoolSize = 10
jppf.datasource.lb.connectionTimeout = 30000
jppf.datasource.lb.idleTimeout = 600000
```

9.4.3.3 Asynchronous persistence

Asynchronous persistence is an asynchronous wrapper for any other load-balancer persistence implementation. The methods of [LoadBalancerPersistence](#) that do not return a result (void return type) are non-blocking and return immediately. All other methods will block until they are executed and their result is available.

The execution of the interface's methods is delegated to a thread pool, whose size can be defined in the configuration or defaults to 1.

This asynchronous persistence can be configured in two forms:

```
# shorten the configuration value for clarity
async = org.jppf.load.balancer.persistence.AsynchronousLoadBalancerPersistence
# asynchronous persistence with default thread pool size
jppf.load.balancing.persistence = ${async} <actual_persistence> <param1> ... <paramN>
# asynchronous persistence with a specified thread pool size
jppf.load.balancer.persistence = ${async} <pool_size> \
    <actual_persistence> <param1> ... <paramN>
```

Here is an example configuration for an asynchronous database persistence:

```
pkg = org.jppf.load.balancer.persistence
# asynchronous database persistence with pool of 4 threads,
# a table named 'JPPF_TEST' and datasource named 'loadBalancerDS'
jppf.load.balancing.persistence = ${pkg}.AsynchronousLoadBalancerPersistence 4 \
    ${pkg}.DatabaseLoadBalancerPersistence JPPF_TEST loadBalancerDS
```

Note: the asynchronous persistence relies, for storage operations, on a queue where elements are uniquely indexed on a (channelID, algorithmID) key. If or when load-balancer states are put in the queue faster than they are persisted, the queue will not grow, because the persistence implementation always replaces entries with an existing key. In other words, the queue size is always bounded by `number_of_channels * number_of_algorithms`. The implication is that you should never see an out of memory condition caused by the asynchronous persistence's footprint, provided the JVM's heap is properly sized.

9.4.4 Managing the persistence store

9.4.4.1 LoadBalancerPersistenceManager interface

Managing the load-balancer persistence store is done via the [LoadBalancerPersistenceManagement](#) interface, which is defined as follows:

```
public interface LoadBalancerPersistenceManagement {
    // Whether load-balancer persistence is enabled
    boolean isPersistenceEnabled();
    // List all the channels that have an entry in the persistence store
    List<String> listAllChannels() throws LoadBalancerPersistenceException;
    // List all algorithms for which a channel has an entry in the persistence store
    List<String> listAlgorithms(String channelID) throws LoadBalancerPersistenceException;
    // List all channels that have an entry for the specified algorithm
    List<String> listAllChannelsWithAlgorithm(String algorithm)
        throws LoadBalancerPersistenceException;
    // Determine whether a channel has an entry for the specified algorithm
    boolean hasAlgorithm(String channelID, String algorithm)
        throws LoadBalancerPersistenceException;
    // Delete all entries in the persistence store
    void deleteAll() throws LoadBalancerPersistenceException;
    // Delete all entries for the specified channel
    void deleteChannel(String channelID) throws LoadBalancerPersistenceException;
    // Delete the specified algorithm state from all the channels that have it
    void deleteAlgorithm(String algorithm) throws LoadBalancerPersistenceException;
    // Delete the specified algorithm state from the specified channel
    void delete(String channelID, String algorithm)
        throws LoadBalancerPersistenceException;
}
```

This interface is persistence implementation-agnostic and is designed to work the same way no matter which persistence is configured.

Note: all "algorithm" parameters used in the methods of [LoadBalancerPersistenceManagement](#) are the real algorithm names, such as "proportional", "rl2", "autotuned",

Here is an example usage:

```
LoadBalancerPersistenceManagement mgt = ...;
try {
    // list all nodes with an entry in the persistence store
    List<String> nodeIDs = mgt.listAllChannels();
    for (String nodeID: nodeIDs) {
        // if the node has an entry for the "proportional" algorithm, delete it
        if (mgt.hasAlgorithm("proportional")) {
            mgt.delete(nodeID, "proportional");
        }
    }
    // ensure there is no longer any entry for the "proportional" algorithm
    assert mgt.listAllChannelsWithAlgorithm("proportional").isEmpty();
} catch (LoadBalancerPersistenceException lbpe) {
    lbpe.printStackTrace();
}
```

9.4.4.2 Managing client-side load-balancing persistence

For client-side load-balancing, the persistence store management interface can be obtained directly via the [JPPFClient](#)'s [getLoadBalancerPersistenceManagement\(\)](#) method, for example:

```
JPPFClient client = new JPPFClient();
LoadBalancerPersistenceManagement mgt = client.getLoadBalancerPersistenceManagement();
// list all driver connections with an entry in the persistence store
List<String> channelIDs = mgt.listAllChannels();
// do something with the list ...
```

9.4.4.3 Managing driver-side load-balancing persistence

To manage its load-balancer persistence store, the JPPF driver provides a management MBean that can be used via standard JMX APIs. The MBean interface is [LoadBalancerPersistenceManagerMBean](#), defined as follows:

```

public interface LoadBalancerPersistenceManagerMBean
extends LoadBalancerPersistenceManagement {
    // The object name under which this MBean is registered with the MBean server
    String MBEAN_NAME = "org.jppf:name=loadBalancerPersistenceManager,type=driver";
}

```

As we can see, it merely extends [LoadBalancerPersistenceManagement](#), adding a constant for its [object name](#). The JPPF driver management APIs provide an easy way to access it with the [JMXDriverConnectionWrapper](#) class:

via a JPPF client:

```

JPPFClient client = new JPPFClient();
JMXDriverConnectionWrapper jmx =
    client.awaitWorkingConnectionPool.awaitWorkingJMXConnection();
LoadBalancerPersistenceManagement mgt = jmx.getLoadBalancerPersistenceManagement();

```

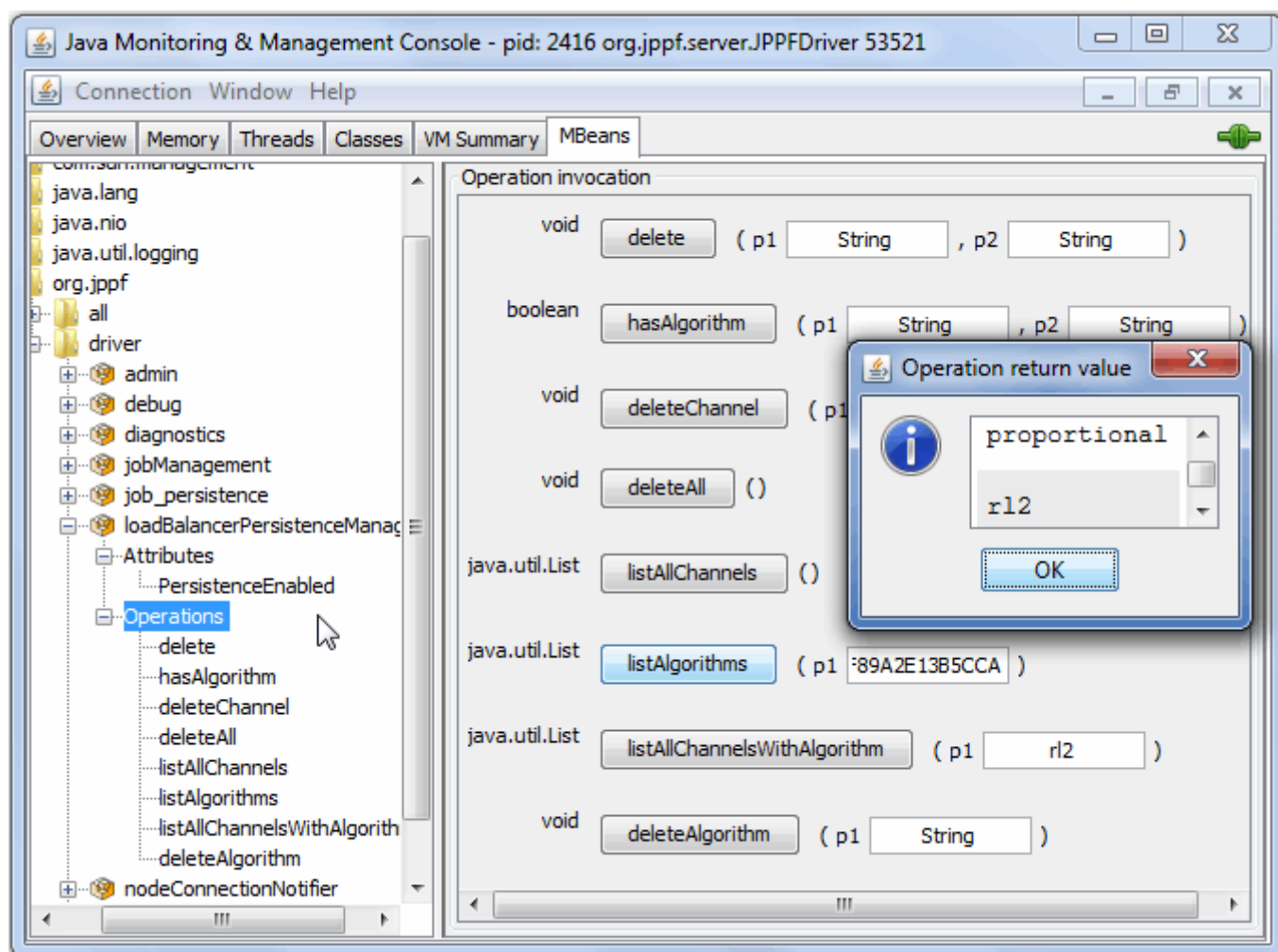
or via direct instantiation:

```

JMXDriverConnectionWrapper jmx =
    new JMXDriverConnectionWrapper("www.myhost.com", 11098, false);
LoadBalancerPersistenceManagement mgt = jmx.getLoadBalancerPersistenceManagement();

```

This MBean is also directly usable via JMX-based tools such as JConsole or VisualVM:



10 Database services

A great many JPPF-based applications have a need to access one or more databases in the computations they submit to the grid. While it is quite easy to define and use database connectivity from within JPPF tasks, we have found that it can also be a source of headaches when addressing issues such as:

- caching and maintaining data sources reusable accross jobs submissions
- handling the availability of the JDBC driver and data source implementation classes
- avoiding the configuration and deployment burden which occurs when database access is needed on a large number of nodes in the grid

To address these issues, JPPF provides a database services component which includes:

- a facility to easily define and configure JDBC data sources from the JPPF configuration (static definition)
- a data source registry and factory to lookup and manipulate data sources, as well as create new ones dynamically.
- a set of functionalities to define data sources in a JPPF driver and propagate them automatically to the nodes

The data source and connection pooling implementations are provided by the [HikariCP](#) library. Consequently, the names of the corresponding configuration properties are those provided in the [HikariCP documentation](#).

10.1 Configuring JDBC data sources

10.1.1 Data source definition

A data source can be defined in the JPPF configuration with a set of properties in the following format:

```
jppf.datasource.<config_id>.name = <datasource_name>
jppf.datasource.<config_id>.<property1> = <value1>
...
jppf.datasource.<config_id>.<propertyN> = <valueN>
```

where:

- *<config_id>* is an arbitrary string used to identify which properties are used to define the data source and to distinguish these properties from those of other data source definitions
- *<datasource_name>* is a user-defined name that will be used to retrieve the data source via API
- *<property_i>* is the name of a [HikariCP-supported property](#)

Here is a concrete example, defining two data sources named "test1DS" and "test2DS", connecting to two distinct MySQL databases and specifying different connection pool settings:

```
# definition of data source "test1DS"
jppf.datasource.test1.name = test1DS
jppf.datasource.test1.driverClassName = com.mysql.jdbc.Driver
jppf.datasource.test1.jdbcUrl = jdbc:mysql://192.168.1.12:3306/test1_db
jppf.datasource.test1.username = testjppf
jppf.datasource.test1.password = mypassword
jppf.datasource.test1.minimumIdle = 5
jppf.datasource.test1.maximumPoolSize = 10
jppf.datasource.test1.connectionTimeout = 30000
jppf.datasource.test1.idleTimeout = 600000

# definition of data source "test2DS"
jppf.datasource.test2.name = test2DS
jppf.datasource.test2.driverClassName = com.mysql.jdbc.Driver
jppf.datasource.test2.jdbcUrl = jdbc:mysql://192.168.1.24:3306/test2_db
jppf.datasource.test2.username = testjppf
jppf.datasource.test2.password = mypassword
jppf.datasource.test2.minimumIdle = 1
jppf.datasource.test2.maximumPoolSize = 20
jppf.datasource.test2.connectionTimeout = 30000
jppf.datasource.test2.idleTimeout = 3000000
```

TIP: to disable a data source definition, simply remove or comment out the "name" property by placing a '#' character at the beginning of the line.

10.1.2 Scoped data sources and propagation to the nodes

When a data source is defined in a JPPF driver configuration, it is possible to specify that its definition should be propagated to the nodes connected to this driver, including new nodes that connect well after the driver is started. To achieve this, you can add a property named "scope" to the data source definition, which has three possible values:

- "local" means the data source definition is intended for the current local JVM only
- "remote" means the definition is to be sent over to the nodes but not used locally
- "any" means the definition will be used both locally by the driver and in the nodes

When the scope is unspecified, it always defaults to "local". Here is an example scoped data source definition:

```
jppf.datasource.node.name = nodeDS
# "nodeDS" data source available in the nodes only
jppf.datasource.node.scope = remote
jppf.datasource.node.driverClassName = com.mysql.jdbc.Driver
jppf.datasource.node.jdbcUrl = jdbc:mysql://192.168.1.12:3306/test1_db
jppf.datasource.node.username = testjppf
jppf.datasource.node.password = mypassword

jppf.datasource.common.name = commonDS
# "commonDS" data source available in the driver and in the nodes
jppf.datasource.common.scope = any
jppf.datasource.common.driverClassName = com.mysql.jdbc.Driver
jppf.datasource.common.jdbcUrl = jdbc:mysql://192.168.1.12:3306/test1_db
jppf.datasource.common.username = testjppf
jppf.datasource.common.password = mypassword
```

Note: the value of the "scope" property is case-insensitive.

10.1.3 Data source execution policy filter

When the configuration of a driver specifies data sources intended for the nodes, with a scope of either "remote" or "any", it is also possible to specify which nodes the data sources are propagated to, by using an execution policy in XML format.

The execution policy is specified using a configuration property named "policy", in this format:

```
jppf.datasource.<config_id>.policy = xml_source_type | xml_source
```

The `xml_source_type` part of the value specifies where to read the XML policy from, and the meaning of `xml_source` depends on its value. The value of `xml_source_type` can be one of:

- "inline": `xml_source` is the actual XML policy specified inline in the configuration
- "file": `xml_source` represents a path, in either the file system or classpath, to an XML file or resource. The path is looked up first in the file system, then in the classpath if it is not present in the file system
- "url": `xml_source` represents a URL to an XML file, including but not limited to, http, https, ftp and file urls.

Note that `xml_source_type` can be omitted, in which case it defaults to "inline".

Here is an example specifying an inline policy:

```
jppf.datasource.node1.name = nodeDS1
jppf.datasource.node1.scope = remote
# ... other properties ...
jppf.datasource.node1.policy = inline | \
<jppf:ExecutionPolicy> \
  <OR> \
    <Equal valueType="string" ignoreCase="false"> \
      <Property>custom.prop</Property> \
      <Value>node1</Value> \
    </Equal> \
    <IsInIPv4Subnet> \
      <Subnet>192.168.1.10-50</Subnet> \
    </IsInIPv4Subnet> \
  </OR> \
</jppf:ExecutionPolicy>
```


TIP: the XML is equivalent to the string produced by invoking the following [ExecutionPolicy API](#):

```
String xml = new Equal("custom.prop", false, "node1")
    .or(new IsInIPv4Subnet("192.168.1.10-50")).toXML();
```

This is much less cumbersome and can be reused by [scripting](#) the value of the "policy" property:

```
# scripting the value with the default Javascript engine
jppf.datasource.node1.policy = inline | $script{ \
    new org.jppf.node.policy.Equal("custom.prop", false, "node1") \
    .or(new org.jppf.node.policy.IsInIPv4Subnet("192.168.1.10-50")).toXML(); \
}$
```

An example using a path to an XML file:

```
jppf.datasource.node2.name = nodeDS2
jppf.datasource.node2.scope = remote
# ... other properties ...
jppf.datasource.node2.policy = file | /home/some_user/jppf/my_policy.xml
```

Using the same XML file specified as a file: url:

```
jppf.datasource.node3.name = nodeDS3
jppf.datasource.node3.scope = remote
# ... other properties ...
jppf.datasource.node3.policy = url | file:///home/some_user/jppf/my_policy.xml
```

10.1.4 Organizing and maintaining data source definitions

As we have seen in previous sections, it is possible to configure data sources with a lot of flexibility. However, especially when you have to deal with multiple data sources, this may tend to bloat the configuration file, introduce duplication of values for the properties, and cause an additional maintenance burden. We strongly encourage you to use the features described in the JPPF configuration guide: [includes, substitutions and scripted values](#).

For instance, let's imagine a scenario where we define two data sources using distinct databases on the same MySQL server. The two data sources will share the same JDBC driver and server URL, which we can put in separate properties referenced in the data source definitions. Furthermore, to avoid bloating the driver's configuration file, we will put these definitions in a separate file in "config/datasource.properties". Here's what the content of "datasource.properties" would look like:

```
# variables referenced in the data source definitions
mysql.driver = com.mysql.jdbc.Driver
mysql.url.base = jdbc:mysql://192.168.1.12:3306
my.scope = remote

# data source "DS1"
jppf.datasource.ds1.name = DS1
jppf.datasource.ds1.scope = ${my.scope}
jppf.datasource.ds1.driverClassName = ${mysql.driver}
jppf.datasource.ds1.jdbcUrl = ${mysql.url.base}/test_db1
jppf.datasource.ds1.username = user1
jppf.datasource.ds1.password = password1

# data source "DS2"
jppf.datasource.ds2.name = DS2
jppf.datasource.ds2.scope = ${my.scope}
jppf.datasource.ds2.driverClassName = ${mysql.driver}
jppf.datasource.ds2.jdbcUrl = ${mysql.url.base}/test_db2
jppf.datasource.ds2.username = user2
jppf.datasource.ds2.password = password2
```

Then, in the main driver configuration file, we would just include "datasource.properties" like this:

```
# include the data source definitions file
#!include file config/datasource.properties
```

We have also seen that we can use scripted property values. This can be especially useful if you do not want to leave clear-text passwords in your configuration files. You can then use a script to invoke an API that will decrypt an encrypted password provided in the configuration. For instance, let's say we define the following method in the class `test.Crypto`:

```
public class Crypto {
```

```
// decrypt the specified string
public static String decrypt(String encrypted) {
    return ...;
}
}
```

We can use it directly in a scripted value for a password:

```
# use an encrypted password in the configuration
jppf.datasource.ds1.password = $script{ test.Crypto.decrypt("Grt!HY+Bp"); }$
```

10.2 The *JPPFDataSourceFactory* API

To access data sources defined in the configuration, use the [JPPFDataSourceFactory](#) class, defined as follows:

```
public final class JPPFDataSourceFactory {
    // Get a data source factory. This method always returns the same instance
    public synchronized static JPPFDataSourceFactory getInstance()

    // Get the data source with the specified name
    public DataSource getDataSource(String name)

    // Get the names of all currently defined data sources
    public List<String> getDataSourceNames()

    // Create a data source from the specified configuration properties
    // This method assumes the property names have no prefix
    public DataSource createDataSource(String name, Properties props)

    // Create one or more data sources from the specified configuration properties.
    // The property names are assumed to be prefixed as in static data source configs
    public Map<String, DataSource> createDataSources(Properties props)

    // Remove the data source with the specified name; also close it and release the
    // resources it is using
    public boolean removeDataSource(String name)

    // Close and remove all the data sources in this registry
    public void clear()
}
```

This class is defined as a JVM-wide singleton, therefore you first need to obtain the singleton instance before using it. For example, to get a defined data source named "DS1", you would do like this:

```
DataSource ds1 = JPPFDataSourceFactory.getInstance().getDataSource("DS1");
```

The methods in [JPPFDataSourceFactory](#) are thread-safe and can be used from multiple threads concurrently, without a need for external synchronization.

10.2.1 Looking up and exploring data sources

Use the **getDataSource(String name)** method to lookup an already defined data source, for instance:

```
DataSource myDS = JPPFDataSourceFactory.getInstance().getDataSource("myDS");
```

To explore the existing data sources, use the **getDataSourceNames()** method, as in this example:

```
JPPFDataSourceFactory factory = JPPFDataSourceFactory.getInstance();
for (String dsName: factory.getDataSourceNames()) {
    DataSource datasource = factory.getDataSource(dsName);
    // do something with the data source ...
}
```

Note: `getDataSource()` and `getDataSourceNames()` only apply to locally-scoped data source definitions. Data sources defined with a "remote" scope will not be retrieved by these methods.

10.2.2 Creating data sources dynamically

[JPPFDataSourceFactory](#) has two methods that allow you to create one or more data sources:

`createDataSource(String name, Properties props)`

Creates a single data source with the specified name and attributes provided in the properties. The names of the properties are not prefixed, meaning that instead of `jppf.datasource.<config_id>.<property_name>` you just use `<property_name>`. Here is an example using the [TypedProperties](#) class for the properties:

```
TypedProperties props = new TypedProperties()
    .setString("driverClassName", "com.mysql.jdbc.Driver")
    .setString("jdbcUrl", "jdbc:mysql://localhost:3306/testjppf")
    .setString("username", "testjppf")
    .setString("password", "testjppf")
    .setInt("maximumPoolSize", 10);

JPPFDataSourceFactory factory = JPPFDataSourceFactory.getInstance();
DataSource ds1 = factory.createDataSource("DS1", props);
```

`createDataSources(Properties props)`

This method enables the creation of one or more data sources at once, using properties with prefixed names. It returns a map associating the created data sources names with the corresponding [DataSource](#) objects. The name of each data source is specified as a prefixed "name" property in the form `jppf.datasource.<config_id>.name = <data_source_name>`. Here is an example defining two data sources named "DS1" and "DS2":

```
TypedProperties props = new TypedProperties();

// property name prefix for "config1" config id
String prefix1 = "jppf.datasource.config1.";
props.setString(prefix1 + "name", "DS1")
    .setString(prefix1 + "driverClassName", "com.mysql.jdbc.Driver")
    // ... other properties ...
    .setInt(prefix1 + "maximumPoolSize", 10);

// property name prefix for "config2" config id
String prefix2 = "jppf.datasource.config2.";
props.setString(prefix2 + "name", "DS2")
    .setString(prefix2 + "driverClassName", "com.mysql.jdbc.Driver")
    // ... other properties ...
    .setInt(prefix2 + "maximumPoolSize", 10);

JPPFDataSourceFactory factory = JPPFDataSourceFactory.getInstance();
Map<String, DataSource> map = factory.createDataSources(props);
```

Note: the `createXXX()` methods can only be used to create locally-scoped data sources. As a consequence, the "scope" property does not apply to dynamically created data sources and should not be specified. If you specify a "remote" scope, then the data source definition will be ignored.

10.2.3 Data source removal and cleanup

The `removeDataSource(String name)` method removes the data source with the specified name and releases any resource it is using. It returns a boolean value which indicates whether the removal operation succeeded. Example:

```
TypedProperties props = ...;
JPPFDataSourceFactory factory = JPPFDataSourceFactory.getInstance();
DataSource ds1 = factory.createDataSource("myDS", props);
// ... do something with the data source, then remove it ...
if (!factory.removeDataSource("myDS")) {
    System.out.println("could not remove data source 'myDS', please look at the logs");
}
```

Similarly, the `clear()` method removes all data sources defined in the [JPPFDataSourceFactory](#) instance.

10.3 Class loading and classpath considerations

10.3.1 HikariCP and JDBC driver libraries

As a general rule, the HikariCP and JDBC driver libraries should always be co-located in the classpath. This means that, if you have one in a JPPF component's classpath, then the other should be in the same component's classpath. For instance, the JPPF driver distribution includes the HikariCP jar file in its 'lib' folder. This is also where the JDBC driver library should be. As per the JPPF distributed class loading mechanism, this will automatically make these libraries available to any standard JPPF node that connects to the driver, and the classes in these libraries will be automatically downloaded when needed by the node.

Following this rule avoids many potential class loading issues and ClassCastException errors due to classes being loaded by multiple class loaders.

10.3.2 Default classpath

By default, the JPPF driver comes with the HikariCP library in its 'lib' folder. This also is where the JDBC drivers for all the databases you use should go. If you use only standard nodes with remote class loading enabled (the default) then it is sufficient to make these libraries available to the nodes.

If you do not wish to deploy the JDBC driver library in the 'lib' folder, the preferred way to specify its location by is to add one or more `-cp <driver_path>` arguments to the driver's [jppf.jvm.options](#) configuration property, for example:

```
jppf.jvm.options = -server -Xmx1g \  
-cp /opt/MySQL/mysql-connector-java-5.1.10/mysql-connector-java-5.1.10-bin.jar
```

10.3.3 Offline nodes

In an [offline node](#), the remote class loading is completely disabled. Consequently, the HikariCP and JDBC driver libraries must be both explicitly deployed to the node's local classpath. This can be done using the same techniques as described in the above section.

The HikariCP library can be found either:

- in the driver distribution: **JPPF-x.y.z-driver/lib/HikariCP-java7-2.4.11.jar**
- or in the source distribution: **JPPF-x.y.z-full-src/JPPF/lib/HikariCP/HikariCP-java7-2.4.11.jar**

10.3.4 JPPF client applications

A JPPF client will automatically create data sources defined in its configuration. Data sources can also be explicitly added with user code, using the [JPPFDataSourceFactory.createDataSources\(\)](#), as illustrated in the "[creating data sources dynamically](#)" section.

When multiple [JPPFClient](#) instances coexist in the same JVM, care must be taken as to the naming of the datasources. If a datasource name is found multiple times, only the first definition will be taken into account. Subsequent definitions will be ignored.

As for offline nodes, the HikariCP and JDBC driver libraries must be explicitly added to the client JVM's classpath.

11 J2EE Connector

11.1 Overview of the JPPF Resource Adapter

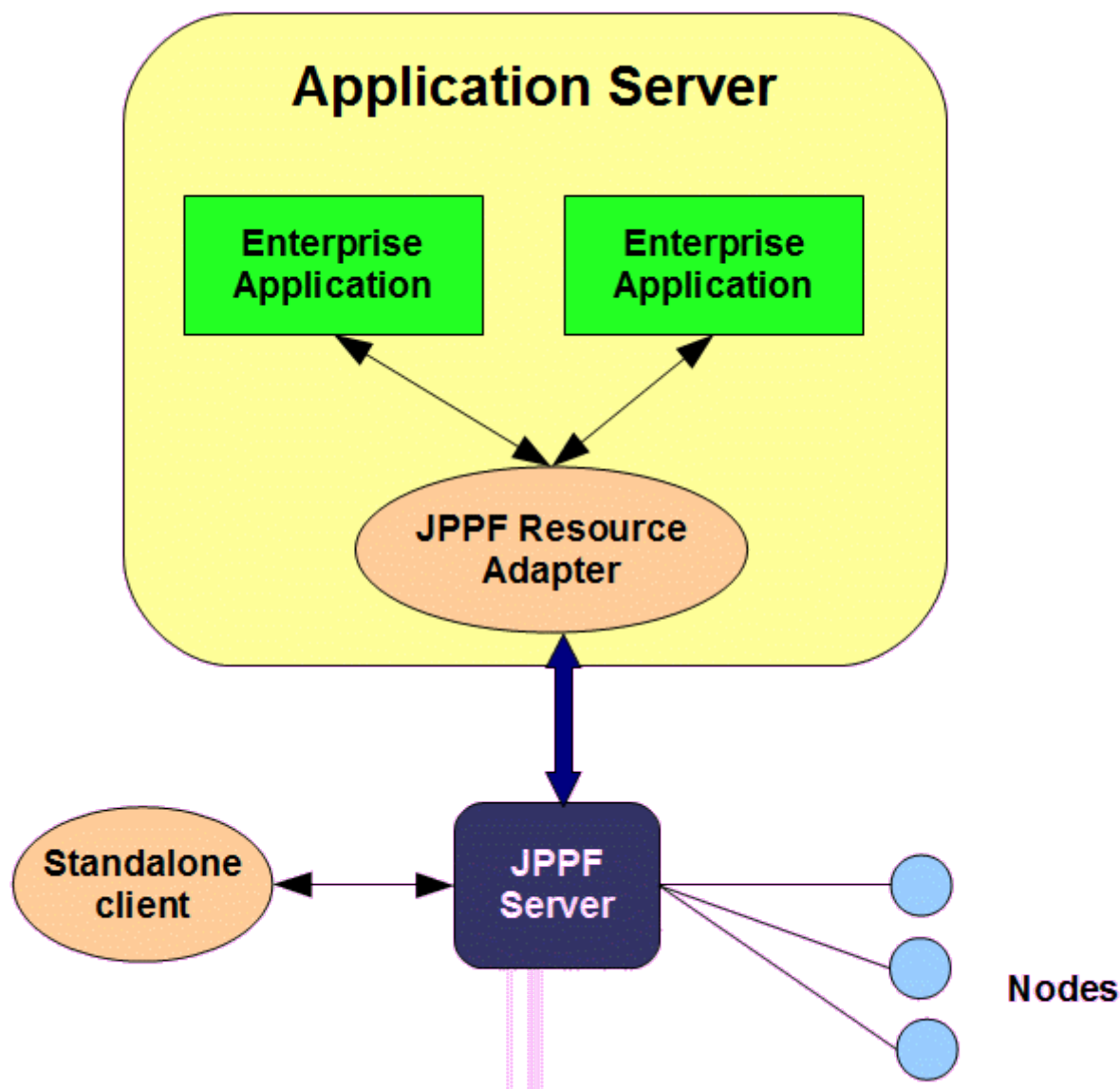
11.1.1 What is it?

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.

11.1.2 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

11.1.3 Architecture



11.2 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 Wildfly 8.x Glassfish 2.x, 3.x IBM Websphere Application Server 8.5+ Oracle Weblogic 9.x, 10.x, 11.x Open Liberty 18.0.0.0 and later

11.3 Configuration and build

11.3.1 Requirement

For building, configuring and customizing the JPPF Resource Adapter, you 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"

11.3.2 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.

11.3.3 Building the JPPF resource adapter

To build the resource adapter:

- Open a command prompt
- Go to the **JPPF-x.y.z-j2ee-connector** folder
- Enter **"ant build"**
- The resulting .rar and .ear files are generated in the **"build"** subfolder.

11.3.4 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	<p>root folder, contains:</p> <ul style="list-style-type: none">– 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: <pre><!-- JPPF Client Configuration --> <config-property> <description>Defines how the JPPF configuration is to be located</description> <!-- 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 --> <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></pre>– the demo application's deployment descriptor 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.

11.4 How to use the connector API

11.4.1 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:

```
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/JCManagedConnectionFactory/eis/JPPFConnectionFactory"**
- on JBoss 4-7: **"java:eis/JPPFConnectionFactory"**
- on Wildfly 8: **"java:/eis/JPPFConnectionFactory"**
- on Websphere: **"java:comp/env/eis/JPPFConnectionFactory"**
- on all other supported servers: **"eis/JPPFConnectionFactory"**

11.4.2 The JPPFConnectionFactory API

In addition to implementing the [ConnectionFactory](#) interface, [JPPFConnectionFactory](#) provides two methods:

```
public class JPPFConnectionFactory implements ConnectionFactory {
    // Determine whether there is a least one working connection to a remote JPPF driver
    public boolean isJPPFDriverAvailable()

    // Enable or disable local (in-JVM) execution of the jobs
    public void enableLocalExecution(final boolean enabled)
}
```

Basically, these methods offer the ability to know whether the J2EE connector is effectively connected to a JPPF driver, and to fallback to local execution when it is not the case.

An example usage of these APIs would be as follows:

```
JPPFConnection connection = null;
try {
    JPPFConnectionFactory factory = ...;
```

```

boolean available = factory.isJPPFDriverAvailable();
// enable local execution, based on the availability of a remote connection
factory.enableLocalExecution(!available);
connection = (JPPFConnection) factory.getConnection();
// ... submit a job ...
} finally {
    if (connection != null) connection.close();
}

```

11.4.3 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:

```

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 connector will resubmit them as soon as it is reset and server connections become available.

11.4.4 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, JobStatusListener 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);
    }
}

```

11.4.5 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.getJobStatus(submitId);
        if (status.equals(JobStatus.COMPLETE)) {
            // if successful process the results
        }
    }
}

```

```

        List<Task<?>> results = connection.getResults(submitID);
    } else if (status.equals(JobStatus.FAILED)) {
        // if failed process the errors
    }
} finally {
    JPPFHelper.closeConnection(connection);
}
}

```

11.4.6 Cancelling a job

The J2EE allows cancelling a job by calling the method [JPPFConnection.cancelJob\(String jobUuid\)](#):

```

public void cancelJob(String jobUuid) throws Exception {
    JPPFConnection connection = null;
    try {
        connection = JPPFHelper.getConnection();
        // cancel the job
        connection.cancelJob(jobUuid);
    } finally {
        JPPFHelper.closeConnection(connection);
    }
}

```

11.4.7 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\(String jobUuid\)](#), 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
        connection.submit(job);
        // wait until the job has completed
        results = connection.awaitResults(job.getUuid());
    } 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.

11.4.8 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 addJobStatusListener(String jobUuid, JobStatusListener listener);

    // Submit a job to the JPPF client and register a status listener
    String submit(JPPFJob job, JobStatusListener listener) throws Exception;
}

```

Note that `submit(JPPFJob, JobStatusListener)` submits the job *and* registers the listener in a single atomic operation. As the job submission is asynchronous, this ensures that no event is missed between the submission of the job and the registration of the listener.

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

```
public interface JobStatusListener extends EventListener {  
    // Called when the status of a job has changed  
    void jobStatusChanged(JobStatusEvent event);  
}
```

Each listener receives events of type [JobStatusEvent](#), defined as follows:

```
public class JobStatusEvent extends EventObject {  
    // get the status of the job  
    public JobStatus getStatus()  
  
    // get the id of the job  
    public String getJobUuid()  
}
```

The possible statuses are defined in the enumerated type [JobStatus](#):

```
public enum JobStatus {  
    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 uuid = connection.submit(job, new JobStatusListener() {
            public void jobStatusChanged(JobStatusEvent event) {
                String uuid = event.getJobUuid();
                JobStatus status = event.getStatus();
                System.out.println("job [" + uuid + "] changed to '" + status + "'");
            }
        });

        // or after the job has been submitted
        connection.addJobStatusListener(id, new JobStatusListener() {
            public void jobStatusChanged(JobStatusEvent event) {
                String uuid = event.getJobUuid();
                JobStatus status = event.getStatus();
                switch(status) {
                    case COMPLETE:// process successful completion
                        break;
                    case FAILED:// process failure
                        break;
                    default:
                        System.out.println("job [" + uuid + "] changed to '" + status + "'");
                        break;
                }
            }
        });
        List<Task<?>> results = connection.awaitResults(uuid);
        // ... process the results ...
    } finally {
        JPPFHelper.closeConnection(connection);
    }
}
```

11.5 Deployment on a J2EE application server

11.5.1 Deployment on JBoss 4.x - 6.x

11.5.1.1 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")

11.5.1.2 Creating a connection factory

Create, in the <JBOSS_HOME>/server/<your_server>/deploy folder, a file named jppf-ra-JBoss-ds.xml. Edit this file with a text editor and add this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
  <no-tx-connection-factory>
    <jndi-name>eis/JPPFConnectionFactory</jndi-name>
    <application-managed-security/>
    <rar-name>jppf_ra_JBoss-rar</rar-name>
    <connection-definition>javax.resource.cci.ConnectionFactory</connection-definition>
    <adapter-display-name>JPPF</adapter-display-name>
    <min-pool-size>0</min-pool-size>
    <max-pool-size>10</max-pool-size>
    <blocking-timeout-millis>50000</blocking-timeout-millis>
    <idle-timeout-minutes>15</idle-timeout-minutes>
  </no-tx-connection-factory>
</connection-factories>
```

You can also [download this file](#).

11.5.1.3 Deploying the demo application

Copy the file "JPPF_J2EE_Demo_JBoss-4.0.ear" in the <JBOSS_HOME>/server/<your_server>/deploy folder

11.5.2 Deployment on JBoss 7+ / Wildfly 8.x

11.5.2.1 Deploying the resource adapter

- in the JBoss administration console, go to the "Deployments" view
- remove any existing JPPF deployments (.rar and .ear)

Deployments

Add Content

Name	Runtime Name	Enabled	En/Disable	Remove
No items!				

1-1 of 0

- click on "Add Content"

- browse to the file "jppf_ra_JBoss-7.rar"

Upload ✕

Step 1/2: Deployment Selection

Please choose a file that you want to deploy.

ild\jppf_ra_JBoss-7.rar

- click "next"

Upload ✕

Step 2/2: Verify Deployment Names

Key: Cn0TGRcXtbOgUMHCA09CusCBzVw=

Name:


Runtime Name:

- 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		<input type="button" value="Enable"/>	<input type="button" value="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		<input type="button" value="Disable"/>	<input type="button" value="Remove"/>

1-1 of 1

11.5.2.2 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:

Deployments

Deployments

Name	Runtime Name	Enabled	En/Disable	Remove
JPPF_J2EE_Demo_JBoss-7.ear	JPPF_J2EE_Demo_JBoss-7.ear		<input type="button" value="Enable"/>	<input type="button" value="Remove"/>
jppf_ra_JBoss-7.rar	jppf_ra_JBoss-7.rar		<input type="button" value="Disable"/>	<input type="button" value="Remove"/>

1-2 of 2

- click on the "Enable" button to start it:

Deployments

Deployments

Name	Runtime Name	Enabled	En/Disable	Remove
JPPF_J2EE_Demo_JBoss-7.ear	JPPF_J2EE_Demo_JBoss-7.ear		<input type="button" value="Disable"/>	<input type="button" value="Remove"/>
jppf_ra_JBoss-7.rar	jppf_ra_JBoss-7.rar		<input type="button" value="Disable"/>	<input type="button" value="Remove"/>

1-2 of 2

11.5.3 Deployment on SunAS / Glassfish

11.5.3.1 Deploying the Resource Adapter

- in Sun AS console, go to "Applications > Connector modules"

The screenshot shows the Sun Java System Application Server Admin Console. The top navigation bar includes links for HOME, VERSION, UPGRADE, REGISTRATION, LOGOUT, and HELP. The user is logged in as 'admin' on 'localhost' in 'domain1'. The main title is 'Sun Java™ System Application Server Admin Console'. The left sidebar shows a tree view with 'Connector Modules' selected. The main content area is titled 'Connector Modules' and contains a description: 'A connector module is used to connect to an Enterprise Information System (EIS) and is packaged in a RAR (Resource Adapter Archive) file or directory.' Below this is a section 'Deployed Connector Modules (0)' with buttons for 'Deploy...', 'Undeploy', 'Enable', and 'Disable'. A table with columns 'Application Name', 'Enabled', 'Location', and 'Actions' is shown, with a message: 'No applications found. Click "Deploy..." above to deploy a new application.'

- click on "Deploy"
- step 1: browse to the "jppf_ra_Glassfish.rar" file

The screenshot shows the 'Deploy Connector Module (Step 1 of 2)' dialog in the Sun Java System Application Server Admin Console. The dialog has 'Next' and 'Cancel' buttons. It contains the text: 'Specify the location of an application to deploy. Applications can be in packaged files, such as .rar, or in the standard Connector Module directory format.' Under the 'Location:' label, there are two radio buttons. The first is selected and labeled 'Package file to be uploaded to the Application Server.' Below it is a text field 'File To Upload:' containing 'aces\SourceForge\jca-client\build\jppf_ra_SunAS-9.0.rar' and a 'Browse...' button. The second radio button is labeled 'Package file or a directory path that is accessible from the server.' Below it is a text field 'File Or Directory:'.

- click "next"
- step 2: leave all default settings and click "Finish"



Common Tasks

Application Server

Applications

Enterprise Applications

Web Applications

EJB Modules

Connector Modules

Lifecycle Modules

App Client Modules

Web Services

Custom MBeans

Resources

Configuration

Application Server > Applications > Connector Modules

Deploy Connector Module (Step 2 of 2)

Finish

Cancel

Specify settings for the connector module you want to deploy. If the module has already been deployed, choose a different application name and deploy it under a new name.

* Indicates required field

General

File Name: jppf_ra_SunAS-9.0.rar

* Application Name: jppf_ra_SunAS-9

Name must contain only alphanumeric, underscore, dash, or dot characters

Thread Pool ID:

Thread pool ID for connector module (resource adapter)

Status:

☒ Enabled

Run:

☐ Verifier

Perform detailed verification before deploying

Registry Type:

None

Description:

Add a description of the component

Resource Adapter Properties

Additional Properties (4)

Name	Value
ServerHost	<input type="text"/>
ConnectionPoolSize	<input type="text"/>
ClassServerPort	<input type="text"/>
AppServerPort	<input type="text"/>

11.5.3.2 Creating a connector connection pool

- in the console tree on the left, go to "Resources > Connectors > Connector Connection Pools"



HOME VERSION UPGRADE REGISTRATION LOGOUT HELP

User: admin Server: localhost Domain: domain1

Sun Java™ System Application Server Admin Console

Application Server > Resources > Connectors > Connector Connection Pools

Connector Connection Pools

Deploy the connector module before creating the pool.

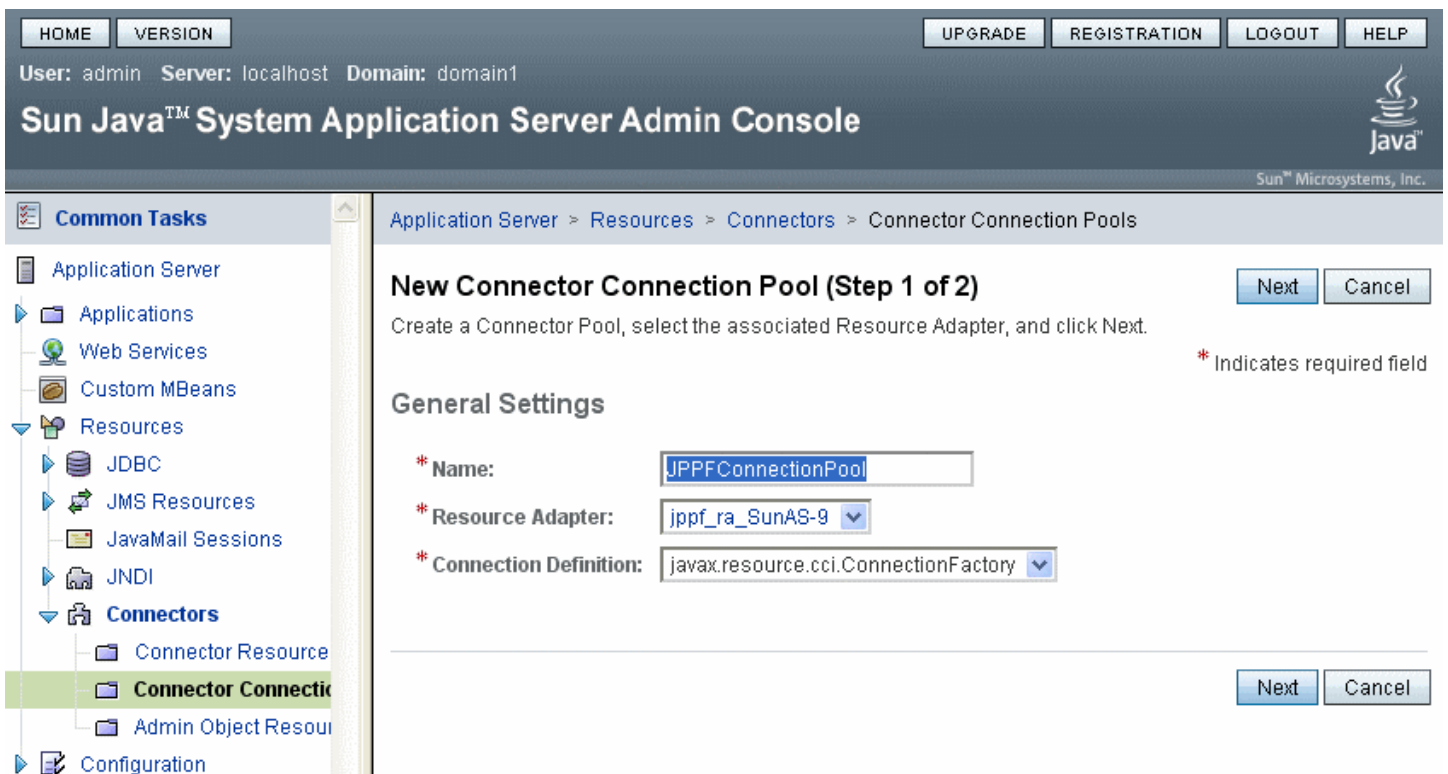
Pools (0)

New... Delete

Name	Description	Resource Adapter
No connection pools found. Click "New..." above to create a connection pool.		

- click on "New"

- step 1: enter "JPPFConnectionPool" as the connection pool name and select "jppf_ra_Glassfish" for the resource adapter



HOME VERSION UPGRADE REGISTRATION LOGOUT HELP

User: admin Server: localhost Domain: domain1

Sun Java™ System Application Server Admin Console

Application Server > Resources > Connectors > Connector Connection Pools

New Connector Connection Pool (Step 1 of 2)

Create a Connector Pool, select the associated Resource Adapter, and click Next.

Next Cancel

* Indicates required field

General Settings

* Name: JPPFConnectionPool

* Resource Adapter: jppf_ra_SunAS-9

* Connection Definition: javax.resource.cci.ConnectionFactory

Next Cancel

- click "next"

- step 2: set the pool parameters, select "NoTransaction" for transaction support

HOME

VERSION

UPGRADE

REGISTRATION

LOGOUT

HELP

User: admin Server: localhost Domain: domain1

Sun™ Microsystems, Inc.

Common Tasks

Application Server

Applications

Web Services

Custom MBeans

Resources

JDBC

JMS Resources

JavaMail Sessions

JNDI

Connectors

Connector Resource

Connector Connection Pools

Admin Object Resources

Configuration

Application Server > Resources > Connectors > Connector Connection Pools

New Connector Connection Pool (Step 2 of 2)

Previous

Finish

Cancel

Verify the Connection Pool settings, add properties defining the value for each property, and click Finish.

General Settings

Name:

JPPFConnectionPool

Resource Adapter:

jppf_ra_SunAS-9

Connection Definition:

javax.resource.cci.ConnectionFactory

Description:

Pool Settings

Initial and Minimum Pool Size:

1

Connections

Maximum Pool Size:

10

Connections

Pool Resize Quantity:

2

Connections

Idle Timeout:

300

Seconds

Max Wait Time:

60000

Milliseconds

On Any Failure:

☐ Close All Connections

Transaction Support:

NoTransaction

Level of transaction support

Connection Validation:

☐ Enabled

Validate connections before passing to application

Properties

Additional Properties (0)

Add Property

Delete Properties

Name	Value
No properties found. Click "Add Property" above to add a property.	

- click "Finish"

11.5.3.3 Creating a connection factory

- in the console tree on the left, go to "Resources > Connectors > Connector Resources"

HOME VERSION UPGRADE REGISTRATION LOGOUT HELP

User: admin Server: localhost Domain: domain1

Sun Java™ System Application Server Admin Console

Sun Microsystems, Inc.

Common Tasks

- Application Server
- Applications
- Web Services
- Custom MBeans
- Resources
 - JDBC
 - JMS Resources
 - JavaMail Sessions
 - JNDI
 - Connectors**
 - Connector Resources**
 - Connector Connection
 - Admin Object Resource
- Configuration

Application Server > Resources > Connectors > Connector Resources

Connector Resources

A connector resource is a program object that provides an application with a connection to an Enterprise Information System (EIS).

Resources (0)

New... Delete

JNDI Name	Enabled	Description
No connector resources found. Click "New..." above to create a connector resource.		

- click on "New"
- for the jndi name, enter "eis/JPPFConnectionFactory"
- for the connection pool, select "JPPFConnectionPool"

HOME VERSION UPGRADE REGISTRATION LOGOUT HELP

User: admin Server: localhost Domain: domain1

Sun Java™ System Application Server Admin Console

Sun Microsystems, Inc.

Common Tasks

- Application Server
- Applications
- Web Services
- Custom MBeans
- Resources
 - JDBC
 - JMS Resources
 - JavaMail Sessions
 - JNDI
 - Connectors**
 - Connector Resources**
 - Connector Connection
 - Admin Object Resource
- Configuration

Application Server > Resources > Connectors > Connector Resources

New Connector Resource

To create a JDBC resource, specify the connection pool with which it is associated. Multiple JDBC resources can use a single connection pool.

* Indicates required field

* JNDI Name: A unique identifier; contain only alphanumeric, underscore, dash, or dot characters

* Pool Name: Use the [Connector Connection Pools](#) page to create new pools

Description:

Status: ☒ Enabled

OK Cancel

- click "OK"
- restart the application server

11.5.3.4 Deploying the demo application

- in SunAS console, go to "Applications > Enterprise Applications"

The screenshot shows the Sun Java System Application Server Admin Console. The top navigation bar includes links for HOME, VERSION, UPGRADE, REGISTRATION, LOGOUT, and HELP. The user is logged in as 'admin' on 'localhost' in the 'domain1' domain. The main title is 'Sun Java™ System Application Server Admin Console'. The left sidebar shows a tree view with 'Common Tasks' and 'Applications' expanded, with 'Enterprise Applications' selected. The main content area shows the 'Enterprise Applications' page with a breadcrumb 'Application Server > Applications > Enterprise Applications'. It includes a description: 'An enterprise application is a J2EE application in an EAR (Enterprise Application Archive) file or directory.' Below this is a section 'Deployed Enterprise Applications (0)' with buttons for 'Deploy...', 'Undeploy', 'Enable', and 'Disable'. A table with columns 'Application Name', 'Enabled', 'Location', and 'Actions' is shown, with a message 'No applications found. Click "Deploy..." above to deploy a new application.'

- click on "Deploy"
- step 1: browse to the file "JPPF_J2EE_Demo_Glassfish.ear"

The screenshot shows the 'Deploy Enterprise Application (Step 1 of 2)' dialog in the Sun Java System Application Server Admin Console. The top navigation bar and user information are the same as the previous screenshot. The left sidebar shows the 'Enterprise Applications' page. The main content area shows the 'Deploy Enterprise Application (Step 1 of 2)' dialog with 'Next' and 'Cancel' buttons. It includes a description: 'Specify the location of an application to deploy. Applications can be in packaged files, such as .ear, or in the standard Enterprise Application directory format.' Below this is a 'Location:' section with two radio buttons. The first radio button is selected and labeled 'Package file to be uploaded to the Application Server.' It has a 'File To Upload:' text field with the value 'forge\jca-client\build\JPPF_J2EE_Demo_SunAS-9.0.ear' and a 'Browse...' button. The second radio button is labeled 'Package file or a directory path that is accessible from the server.' It has a 'File Or Directory:' text field.

- click "next"
- step 2: leave all default values

HOMEVERSION

UPGRADEREGISTRATIONLOGOUTHELP

User: admin Server: localhost Domain: domain1

Sun Java™ System Application Server Admin Console

Sun™ Microsystems, Inc.

Common Tasks

Application Server

Applications

Enterprise Applications

Web Applications

EJB Modules

Connector Modules

Lifecycle Modules

App Client Modules

Web Services

Custom MBeans

Resources

Configuration

Application Server > Applications > Enterprise Applications

Deploy Enterprise Application (Step 2 of 2)

FinishCancel

By default, the module is available as soon as it is deployed. To disable the module so that it 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: ☐ RMISubs
Generate static RMI stubs and put in client.jar

- click "Finish"
- restart the application server

11.5.4 Deployment on Websphere

11.5.4.1 Deploying the Resource Adapter

- in WAS console, go to "Resources > Resource Adapters > Resource adapters"
- select scope = "Node"

Integrated Solutions Console Welcome lolo Help | Logout

Resource adapters Close page

View: All tasks

- Welcome
- Guided Activities
- Servers
- Applications
- Resources
 - Schedulers
 - Object pool managers
 - JMS
 - JDBC
 - Resource Adapters
 - Resource adapters
 - J2C connection factories
 - J2C activation specifications
 - J2C administered objects
 - Asynchronous beans
 - Cache instances
 - Mail
 - URL
 - Resource Environment
- Security
- Environment
- System administration

Resource adapters

Use this page to manage resource adapters, which provide the fundamental interface for connecting applications to an Enterprise Information System (EIS). The WebSphere(R) Relational Resource Adapter is embedded within the product to provide access to relational databases. To access another type of EIS, use this page to install a standalone resource adapter archive (RAR) file. You can configure multiple resource adapters for each installed RAR file.

Scope: Cell=**biglolo2Node01**Cell, Node=**biglolo2Node01**

Scope specifies the level at which the resource definition is visible. For detailed information on what scope is and how it works, [see the scope settings help](#)

Node=biglolo2Node01

Preferences

Install RAR New Delete

Select	Name	Scope
None		
Total 0		

Field help
For field help information, select a field label or list marker when the help cursor appears.

Page help
[More information about this page](#)

Command Assistance
[View administrative scripting command for last action](#)

- click "Install RAR"

- in the "Install RAR file" page, browse to the "jppf_ra_WebSphere.rar" file

The screenshot shows the 'Install RAR File' dialog box within the Integrated Solutions Console. The left sidebar contains a navigation tree with categories like 'Welcome', 'Guided Activities', 'Servers', 'Applications', 'Resources', 'Security', and 'Environment'. The 'Resources' category is expanded, showing sub-items like 'Schedulers', 'Object pool managers', 'JMS', 'JDBC', 'Resource Adapters', 'Asynchronous beans', 'Cache instances', 'Mail', 'URL', and 'Resource Environment'. The 'Resource Adapters' sub-item is selected. The main area of the dialog is titled 'Install RAR File' and contains instructions: 'Use this page to install a RAR file in one of two ways. You can either upload a RAR file from the local file system, or specify an existing RAR file on a server. The RAR file must be installed at the node level, and you can select the node below.' Below the instructions, there are two sections: 'Path' and 'Scope'. The 'Path' section has two radio buttons: 'Local path:' (selected) and 'Server path:'. The 'Local path:' section has a text field 'Specify path' containing 'D:\Workspaces\SourceForge\jca-client\build\jppf_ra_WAS-6.1.rar' and a 'Browse...' button. The 'Server path:' section has a text field 'Specify path'. The 'Scope' section has a dropdown menu 'Node' with 'biglolo2Node01' selected. At the bottom of the dialog are 'Next' and 'Cancel' buttons.

- click "Next"
- click "OK"
- click "Save directly to the master configuration"

The screenshot shows the 'Resource adapters' page in the Integrated Solutions Console. The left sidebar is the same as in the previous screenshot. The main area is titled 'Resource adapters' and contains a description: 'Use this page to manage resource adapters, which provide the fundamental interface for connecting applications to an Enterprise Information System (EIS). The WebSphere(R) Relational Resource Adapter is embedded within the product to provide access to relational databases. To access another type of EIS, use this page to install a standalone resource adapter archive (RAR) file. You can configure multiple resource adapters for each installed RAR file.' Below the description, there is a checkbox 'Scope: Cell=biglolo2Node01Cell, Node=biglolo2Node01' which is checked. Below this is a text field 'Node=biglolo2Node01' with a dropdown arrow. There is a section 'Preferences' with buttons 'Install RAR', 'New', and 'Delete'. Below these are icons for 'Select', 'Copy', 'Paste', and 'Delete'. Below the icons is a table with columns 'Select', 'Name', and 'Scope'. The table has one row with a checkbox, the name 'JPPF', and the scope 'Node=biglolo2Node01'. At the bottom of the table is the text 'Total 1'. On the right side of the page, there is a 'Help' section with 'Field help', 'Page help', and 'Command Assistance' links.

11.5.4.2 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

View: All tasks

Welcome

Guided Activities

Servers

Applications

Resources

Schedulers

Object pool managers

JMS

JDBC

Resource Adapters

Resource adapters

J2C connection factories

J2C activation specifications

J2C administered objects

Asynchronous beans

Cache instances

Mail

URL

Resource Environment

Security

Environment

System administration

Users and Groups

Monitoring and Tuning

Troubleshooting

Service integration

UDDI

Resource adapters

Resource adapters

Messages

Additional Properties for this object will not be available to edit until its general properties are applied by clicking on either Apply or OK.

Resource adapters > JPPF > J2C connection factories > New

Use this page to create a connection factory for use with the resource adapter. The connection factory is a collection of configuration values that define a WebSphere(R) Application Server connection to your Enterprise Information System (EIS). The connection pool manager uses these properties as directions for allocating connections during runtime. You can configure multiple connection factories for each resource adapter.

Configuration

General Properties

* Scope

cells:biglolo2Node01Cell:nodes:biglolo2Node01

* Provider

JPPF

* Name

JPPF Connection Factory

JNDI name

eis/JPPFConnectionFactory

Description

* Connection factory interface

javax.resource.cci.ConnectionFactory

Category

Component-managed authentication alias

Component-managed authentication alias

(none)

The additional properties will not be available until the general properties for this item are applied or saved.

Additional Properties

Connection pool properties

Advanced connection factory properties

Custom properties

Related Items

- click "OK"
- click "Save directly to the master configuration"
- Restart the application server

11.5.4.3 Deploying the demo application

- in Websphere console, go to "Applications > Enterprise Applications"

The screenshot shows the IBM Integrated Solutions Console interface. The top navigation bar includes links for weather, Slashdot, MyCheckFree, java.sun.com, Eclipse, BOA, owa, QualityHost, freshmeat.net, JPPF, and Google. The main header displays "Integrated Solutions Console Welcome lol" and "Enterprise Applications" with a "Close page" link. The left sidebar contains a "View: All tasks" dropdown and a tree of navigation items: Welcome, Guided Activities, Servers, Applications (selected), Resources, Security, Environment, System administration, Users and Groups, Monitoring and Tuning, Troubleshooting, Service integration, and UDDI. The "Enterprise Applications" page title is followed by a "Close page" link. Below the title, the "Enterprise Applications" section explains that this page is used to manage installed applications. A "Preferences" section contains buttons for Start, Stop, Install, Uninstall, Update, Rollout Update, Remove File, Export, and E. Below these are icons for file operations. A table lists installed applications:

Select	Name	Application Status
<input type="checkbox"/>	DefaultApplication	
<input type="checkbox"/>	ivtApp	
<input type="checkbox"/>	query	
Total 3		

- click on "Install"
- browse to the file "JPPF_J2EE_Demo_Websphere.ear"
- select "Prompt me only when additional information is required".

The screenshot shows the "Preparing for the application installation" dialog box in the IBM Integrated Solutions Console. The dialog has a title bar "Preparing for the application installation" and a "Help" button. The main text says "Specify the EAR, WAR, JAR, or SAR module to upload and install." Below this, there are two sections: "Path to the new application" and "Context root". The "Path to the new application" section has two radio buttons: "Local file system" (selected) and "Remote file system". The "Local file system" section has a "Full path" text box containing "PF_J2EE_Demo_WAS-6.1.ear" and a "Browse..." button. The "Remote file system" section has a "Full path" text box and a "Browse..." button. The "Context root" section has a text box and a note: "Used only for standalone Web modules (.war files) and SIP modules (.sar files)". Below these sections is a section titled "How do you want to install the application?" with two radio buttons: "Prompt me only when additional information is required." (selected) and "Show me all installation options and parameters." At the bottom of the dialog are "Next" and "Cancel" buttons. On the right side of the console, there is a "Help" panel with "Field help" (Local file system path) and "Page help" (More information about this page).

- click "Next"
- step 1: click "Next"
- step 2: check module "jppftest"

- View: All tasks
- Welcome
 - Guided Activities
 - Servers
 - Applications
 - Enterprise Applications
 - Install New Application

- Resources
- Security
- Environment
- System administration
- Users and Groups
- Monitoring and Tuning
- Troubleshooting
- Service integration
- UDDI

Enterprise Applications

Close page

Install New Application

Specify options for installing enterprise applications and modules.

Step 1 Select installation options

→ Step 2: Map modules to servers

★ Step 3 Map resource references to resources

★ Step 4 Map virtual hosts for Web modules

Step 5 Summary

Map modules to servers

Specify targets such as application servers or clusters of application servers where you want to install modules. Modules can be installed on the same application server or on different application servers. Also, specify the Web servers as targets that serve as routers for requests. A plug-in configuration file (plugin-cfg.xml) for each Web server is generated, based on the configuration.

Clusters and Servers:

WebSphere:cell=biglolo2Node01Cell,node=biglolo2Node01,server=server1

Apply



Select	Module	URI	Server
<input checked="" type="checkbox"/>	jppftest	jppftest.war,WEB-INF/web.xml	WebSphere:cell=biglolo2Node01Cell,node=biglolo2Node01,server=server1

Previous

Next

Cancel

- click "Next"

- step 3: select "None" for the authentication method

- check the "jppftest" module and enter "eis/JPPFConnectionFactory" as Target Resource JNDI Name

Install New Application

Specify options for installing enterprise applications and modules.

Step 1 Select installation options

Step 2 Map modules to servers

→ Step 3: Map resource references to resources

★ Step 4 Map virtual hosts for Web modules

Step 5 Summary

Map resource references to resources

Each resource reference that is defined in your application must be mapped to a resource.

javax.resource.cci.ConnectionFactory

To modify Resource Authentication method (if Authorization type is 'container'):

- Select one or more checkboxes in the table
- Select either 'none', 'default', or 'custom login configuration'
 - if 'none' is selected:
 - Select one or more checkboxes in the table
 - if 'default' is selected:
 - select an authentication data entry from the dropdown menu
 - Click Apply
 - if 'custom login configuration' is selected:
 - select a custom login configuration from the dropdown menu
 - Click Apply
 - To edit the properties of the custom login configuration, click Mapping Properties in the table

Specify authentication method:

- ☒ None
- ☐ Use default method (many-to-one mapping)

Authentication data entry

Select... ▼

- ☐ Use custom login configuration

Application login configuration

Select... ▼

Apply



Select	Module	EJB	URI	Resource Reference	Target Resource JNDI Name	Login configuration
<input checked="" type="checkbox"/>	jppftest		jppftest.war,WEB-INF/web.xml	eis/JPPFConnectionFactory	eis/JPPFConnectionFactory Browse...	Resource authorization: Per application

Previous

Next

Cancel

- click "Next"
- step 4: check the "jppftest" module and keep "default_host" as the Virtual host

The screenshot shows the 'Install New Application' wizard in the Integrated Solutions Console. The left sidebar contains a navigation tree with 'Enterprise Applications' and 'Install New Application' selected. The main panel is titled 'Map virtual hosts for Web modules' and includes instructions: 'Specify the virtual host where you want to install the Web modules that are contained in your application. You can install Web modules on the same virtual host or disperse them among several hosts.' Below this, there is a checkbox for 'Apply Multiple Mappings' and a table with columns 'Select', 'Web module', and 'Virtual host'. The table contains one row with a checked checkbox, 'jppftest', and 'default_host'. At the bottom are 'Previous', 'Next', and 'Cancel' buttons. A help panel on the right shows 'Field help' and 'Page help' sections.

- click "Next"
- step 5: click "Finish"
- click "Save directly to the master configuration"
- in the list of enterprise applications, check "JPPF Demo"

The screenshot shows the 'Enterprise Applications' page in the Integrated Solutions Console. A message box at the top states: 'Application JPPF Demo on server server1 and node biglolo2Node01 started successfully.' Below this, the 'Enterprise Applications' section provides instructions: 'Use this page to manage installed applications. A single application can be deployed onto multiple servers.' There is a 'Preferences' section with buttons for 'Start', 'Stop', 'Install', 'Uninstall', 'Update', 'Rollout Update', 'Remove File', 'Export', and 'Export D'. Below these buttons is a table with columns 'Select', 'Name', and 'Application Status'. The table lists four applications: 'DefaultApplication', 'JPPF Demo' (which is checked), 'ivtApp', and 'query'. All applications have a green arrow icon in the 'Application Status' column. At the bottom of the table, it says 'Total 4'.

- click on "Start"
- restart the application server

11.5.5 Deployment on Weblogic

11.5.5.1 Deploying the resource adapter

- in Weblogic console, go to "Deployments"
- click on "Lock & Edit"

The screenshot displays the Weblogic Server Administration Console interface. The top navigation bar includes the BEA logo, the title "WEBLOGIC SERVER ADMINISTRATION CONSOLE", and user information "Welcome, lcohen" connected to "wls920_domain". Navigation links for Home, Log Out, Preferences, Help, and AskBEA are present. The left sidebar contains a "Change Center" with a message about pending changes and buttons for "Lock & Edit" and "Release Configuration". Below this is the "Domain Structure" tree, which is expanded to show "wls920_domain" > "Environment" > "Deployments". The "How do I..." section at the bottom left offers links to "Install an Enterprise application" and "Configure an Enterprise application". The main content area, titled "Summary of Deployments", has tabs for "Control" and "Monitoring". A text block explains that the page lists J2EE applications and modules that can be managed. Below this, a "Deployments" section features a table with columns for Name, State, Type, and Deployment Order. The table is currently empty, displaying "Showing 0 - 0 of 0" and "Previous | Next" links. Above and below the table are sets of control buttons: "Install", "Update", "Delete", "Start", and "Stop".

- click "Install"
- navigate to the "jppf_ra_Weblogic.rar" file and select it

WEBLOGIC SERVER
 ADMINISTRATION CONSOLE

Change Center
 Welcome, lcohen
 Connected to: wls920_domain
 Home
 Log Out
 Preferences
 Help
 As

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)

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
<input type="radio"/>	JPPF_J2EE_Demo_JBoss-4.0.ear
<input type="radio"/>	JPPF_J2EE_Demo_Oracle-10.ear
<input type="radio"/>	JPPF_J2EE_Demo_SunAS-9.0.ear
<input type="radio"/>	JPPF_J2EE_Demo_WAS-6.1.ear
<input type="radio"/>	JPPF_J2EE_Demo_Weblogic-9.2.ear
<input type="radio"/>	jppf_ra_JBoss-4.0.rar
<input type="radio"/>	jppf_ra_Oracle-10.rar
<input type="radio"/>	jppf_ra_SunAS-9.0.rar
<input type="radio"/>	jppf_ra_WAS-6.1.rar
<input checked="" type="radio"/>	jppf_ra_Weblogic-9.2.rar

Back Next Finish Cancel

- click "Next"
- click "Next"
- click "Finish"


WEBLOGIC SERVER
 ADMINISTRATION CONSOLE

Change Center

View changes and restarts

Pending changes exist. They must be activated to take effect.

Activate Changes
 Undo All Changes

Domain Structure

wls920_domain

Environment
 Deployments
 Services
 Security Realms
 Interoperability
 Diagnostics

How do I...

Install an Enterprise application
 Configure an Enterprise application
 Update (redeploy) an Enterprise application
 Start and stop a deployed Enterprise application
 Monitor the modules of an Enterprise application
 Deploy EJB modules
 Install a Web application

System Status

Welcome, lcohen
 Connected to: wls920_domain
 Home
 Log Out
 Preferences
 Help
 AskBEA

Home > Summary of Deployments > jppf_ra_Weblogic-9 > Summary of Environment > **Summary of Deployments**

Messages

The deployment has been installed and added to the list of pending changes successfully.
 You must also activate the pending changes to commit this, and other updates, to the active system.

Summary of Deployments


Control
 Monitoring

This page displays a list of J2EE Applications and stand-alone application modules that have been installed to this domain. Installed applications and modules can be started, stopped, updated (redeployed), or deleted from the domain by first selecting the application name and using the controls on this page.

To install a new application or module for deployment to targets in this domain, click the Install button.

Deployments

Install
 Update
 Delete
 Start
 Stop
 Showing 1 - 1 of 1
 Previous
 Next

<input type="checkbox"/>	Name	State	Type	Deployment Order
<input type="checkbox"/>	 jppf_ra_Weblogic-9	distribute Initializing	Resource Adapter	100

Install
 Update
 Delete
 Start
 Stop
 Showing 1 - 1 of 1
 Previous
 Next

- click "Activate Changes"
- in the list of deployments, check "jppf_ra_Weblogic"
- select "Start > Servicing all requests"

WEBLOGIC SERVER
 ADMINISTRATION CONSOLE

Change Center
 View changes and restarts
 Click the Lock & Edit button to modify, add or delete items in this domain.
 Lock & Edit
 Release Configuration

Domain Structure
 wls920_domain
 Environment
Deployments
 Services
 Security Realms
 Interoperability
 Diagnostics

How do I...
 Install an Enterprise application
 Configure an Enterprise application
 Update (redeploy) an Enterprise application
 Start and stop a deployed Enterprise application
 Monitor the modules of an Enterprise application
 Deploy EJB modules
 Install a Web application

Welcome, lcohen Connected to: wls920_domain Home Log Out Preferences Help AskBE

Home > Summary of Deployments > jppf_ra_Weblogic-9 > Summary of Environment > **Summary of Deployments**

Messages
 All changes have been activated. No restarts are necessary.

Summary of Deployments
 Control Monitoring

This page displays a list of J2EE Applications and stand-alone application modules that have been installed to this domain. Installed applications and modules can be started, stopped, updated (redeployed), or deleted from the domain by first selecting the application name and using the controls on this page.
 To install a new application or module for deployment to targets in this domain, click the Install button.

Deployments
 Install Update Delete Start Stop Showing 1 - 1 of 1 Previous | Next

	Name	Type	Deployment Order
<input checked="" type="checkbox"/>	jppf_ra_Weblogic-9	Prepared Resource Adapter	100


 Install Update Delete Start Stop Showing 1 - 1 of 1 Previous | Next

- 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.

11.5.5.2 Deploying the demo application

- in Weblogic console, go to "Deployments"
- click on "Lock & Edit"

**WEBLOGIC SERVER**
ADMINISTRATION CONSOLE

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**

Summary of Deployments

Control **Monitoring**


This page displays a list of J2EE Applications and stand-alone application modules that have been installed to this domain. Installed applications and modules can be started, stopped, updated (redployed), or deleted from the domain by first selecting the application name and using the controls on this page.

To install a new application or module for deployment to targets in this domain, click the Install button.

Deployments

Install Update Delete Start Stop


Showing 1 - 1 of 1 Previous | Next

<input type="checkbox"/>	Name ^	State	Type	Deployment Order
<input type="checkbox"/>	 jppf_ra_Weblogic-9	Active	Resource Adapter	100

Install Update Delete Start Stop

Showing 1 - 1 of 1 Previous | Next

- click "Install"
- navigate to the "JPPF_J2EE_Demo_Weblogic.ear" file and select it

**WEBLOGIC SERVER**
ADMINISTRATION CONSOLE

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
<input type="radio"/>	JPPF_J2EE_Demo_IBoss-4.0.ear
<input type="radio"/>	JPPF_J2EE_Demo_Oracle-10.ear
<input type="radio"/>	JPPF_J2EE_Demo_SunAS-9.0.ear
<input type="radio"/>	JPPF_J2EE_Demo_WAS-6.1.ear
<input checked="" type="radio"/>	JPPF_J2EE_Demo_Weblogic-9.2.ear
<input type="radio"/>	jppf_ra_IBoss-4.0.rar
<input type="radio"/>	jppf_ra_Oracle-10.rar
<input type="radio"/>	jppf_ra_SunAS-9.0.rar
<input type="radio"/>	jppf_ra_WAS-6.1.rar
<input type="radio"/>	jppf_ra_Weblogic-9.2.rar

Back

Next

Finish

Cancel

- click "Next"
- click "Finish"

**WEBLOGIC SERVER**
ADMINISTRATION CONSOLE

Change Center

View changes and restarts

Pending changes exist. They must be activated to take effect.

Activate Changes

Undo All Changes

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

Messages

☒ The deployment has been installed and added to the list of pending changes successfully.

☒ You must also activate the pending changes to commit this, and other updates, to the active system.

Summary of Deployments

Control

Monitoring

This page displays a list of J2EE Applications and stand-alone application modules that have been installed to this domain. Installed applications and modules can be started, stopped, updated (redeployed), or deleted from the domain by first selecting the application name and using the controls on this page.

To install a new application or module for deployment to targets in this domain, click the Install button.

Deployments

Install

Update

Delete

Start

Stop

Showing 1 - 2 of 2 Previous | Next

<input type="checkbox"/>	Name ^	State	Type	Deployment Order
<input type="checkbox"/>	 JPPF_J2EE_Demo_Weblogic-9	distribute Initializing	Enterprise Application	100
<input type="checkbox"/>	 jppf_ra_Weblogic-9	Active	Resource Adapter	100

Install

Update

Delete

Start

Stop

Showing 1 - 2 of 2 Previous | Next

- click "Activate Changes"
- in the list of deployments, check "JPPF_J2EE_Demo_Weblogic"
- select "Start > Servicing All Requests"

WEBLOGIC SERVER
 ADMINISTRATION CONSOLE

Change Center

[View changes and restarts](#)

Click the Lock & Edit button to modify, add or delete items in this 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**

Messages

All changes have been activated. No restarts are necessary.

Summary of Deployments

Control
 Monitoring

This page displays a list of J2EE Applications and stand-alone application modules that have been installed to this domain. Installed applications and modules can be started, stopped, updated (redeployed), or deleted from the domain by first selecting the application name and using the controls on this page.

To install a new application or module for deployment to targets in this domain, click the Install button.

Deployments

Install
 Update
 Delete
 Start
 Stop
 Showing 1 - 2 of 2
 Previous | Next

<input type="checkbox"/>	Name ^		Type	Deployment Order
<input checked="" type="checkbox"/>	JPPF_J2EE_Demo_Weblogic-9	Prepared	Enterprise Application	100
<input type="checkbox"/>	jppf_ra_Weblogic-9	Active	Resource Adapter	100

Install
 Update
 Delete
 Start
 Stop
 Showing 1 - 2 of 2
 Previous | Next

- Click "Yes"
- the state of the demo application must now show as "Active"
- restart the application server

11.5.6 Deployment on Open Liberty

11.5.6.1 Deploying the JPPF resource adapter

1) copy the file `jppf_ra_OpenLiberty.rar` in this folder: `<OPEN_LIBERTY_HOME>/usr/servers/<server>`, where `<OPEN_LIBERTY_HOME>` is the root installation folder for Open Liberty, and `<server>` is the name of the server profile you are using.

2) add the following features to your `server.xml`:

```
<featureManager>
  <feature>jndi-1.0</feature>
  <feature>jca-1.7</feature>
</featureManager>
```

3) In the `server.xml` file, configure the resource adapter as follows:

```
<resourceAdapter id="jppf_ra" location="${server.config.dir}/jppf_ra_OpenLiberty.rar">
  <classloader apiTypeVisibility="spec,ibm-api,api,stable,third-party"/>
</resourceAdapter>
```

4) In the `server.xml` file, configure the JPPF connection factory as follows:

```
<connectionFactory jndiName="eis/JPPFConnectionFactory">
  <properties.jppf_ra ConfigurationSource="classpath|jppf.properties"/>
</connectionFactory>
```

11.5.6.2 Deploying the demo application

1) copy the file `JPPF_J2EE_Demo_OpenLiberty.ear` to `<OPEN_LIBERTY_HOME>/usr/servers/<server>/apps`

2) add the following feature to your `server.xml`:

```
<featureManager>
  <feature>jsp-2.3</feature>
</featureManager>
```

3) configure the enterprise application in your `server.xml`:

```
<application type="ear" id="JPPF_J2EE_Demo_OpenLiberty"
  location="JPPF_J2EE_Demo_OpenLiberty.ear" name="JPPF_J2EE_Demo_OpenLiberty">
  <classloader classProviderRef="jppf_ra"
    apiTypeVisibility="spec,ibm-api,api,stable,third-party"/>
</application>
```

Important: the value of the `classProviderRef` attribute in the class loader declaration of the application, must match exactly the value of the resource adapter's `id` attribute, otherwise the application will not be able to access the JPPF classes.

11.5.6.3 Full working server.xml example

```
<?xml version="1.0" encoding="UTF-8"?>
<server description="JPPF server">
  <featureManager>
    <feature>jndi-1.0</feature>
    <feature>jsp-2.3</feature>
    <feature>jca-1.7</feature>
  </featureManager>

  <httpEndpoint id="defaultHttpEndpoint" host="*" httpPort="9080" httpsPort="9443" />

  <!-- Do not automatically expand WAR files and EAR files -->
  <applicationManager autoExpand="false"/>

  <!-- Deployment of the JPPF demo application -->
  <application type="ear" id="JPPF_J2EE_Demo_OpenLiberty"
    location="JPPF_J2EE_Demo_OpenLiberty.ear" name="JPPF_J2EE_Demo_OpenLiberty">
    <classloader classProviderRef="jppf_ra"
      apiTypeVisibility="spec,ibm-api,api,stable,third-party"/>
  </application>

  <!--
  Deployment of the JPPF resource adapter.
  The .rar file is assumed to be in the same directory as this server.xml
  -->
  <resourceAdapter id="jppf_ra" location="${server.config.dir}/jppf_ra_OpenLiberty.rar">
    <classloader apiTypeVisibility="spec,ibm-api,api,stable,third-party"/>
  </resourceAdapter>

  <!-- Configuration of the JPPF connection factory -->
  <connectionFactory jndiName="eis/JPPFConnectionFactory">
    <properties.jppf_ra ConfigurationSource="classpath|jppf.properties"/>
  </connectionFactory>
</server>
```

11.6 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.

11.7 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-x.y.z-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.z-j2ee-connector/appserver/MyServer-1.0` directory, edit the relevant configuration files and deployment descriptors.
3. Open the `build.xml` build script, in the `JPPF-x.y.z-j2ee-connector` folder, with a text editor.
4. At the start of the file, you will see the following section:

```
<!-- ===== -->
<!-- definition of application server-specific properties -->
<!-- the value is used to generate the names of the corresponding EAR and RAR -->
<!-- ===== -->
<property name="was" value="Websphere"/>
<property name="jboss" value="JBoss"/>
<property name="jboss7" value="JBoss-7"/>
<property name="wildfly8" value="Wildfly-8"/>
<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:

```
<build.ear appserver="${myserver10}"/>
```

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:

```
<rar appserver="${myserver10}"/>
```

You may also remove or comment out those you do not need.

12 Configuration properties reference

12.1 Driver properties

Name	Default value	Description
jppf.discovery.broadcast.exclude.ipv4	null	Prevent broadcast to the specified IPv4 addresses (exclusive filter, server only)
jppf.discovery.broadcast.exclude.ipv6	null	Prevent broadcast to the specified IPv6 addresses (exclusive filter, server only)
jppf.discovery.broadcast.include.ipv4	null	Broadcast to the specified IPv4 addresses (inclusive filter, server only)
jppf.discovery.broadcast.include.ipv6	null	Broadcast to the specified IPv6 addresses (inclusive filter, server only)
jppf.discovery.broadcast.interval	5000	UDP broadcast interval in milliseconds
jppf.discovery.enabled	true	Enable/disable server discovery via UDP multicast
jppf.java.path	null	Full path to the Java executable
jppf.jvm.options	null	JVM options for the node or server process
jppf.load.balancing.algorithm	proportional	Load balancing algorithm name
jppf.load.balancing.persistence	null	Class name of the implementation of a load-balancer persistence in the driver or client, with optional parameters
jppf.load.balancing.persistence.hash	SHA-1	The hash function used to generate load-balancer state identifiers
jppf.load.balancing.profile	jppf	Load balancing parameters profile name
jppf.local.node.bias	true	Whether bias towards local node for scheduling is enabled in the driver
jppf.local.node.enabled	false	Whether to enable a node to run in the same JVM as the driver
jppf.management.enabled	true	Enable/disable management of the node or server
jppf.management.server.forwarder	null	Fully qualified class name of a MBeanServerForwarder implementation with optional space-separated string parameters
jppf.management.ssl.enabled	false	Deprecated: <i>management is now enabled on both plain and secure connections via "jppf.management.enabled"</i> Enable/disable JMX via secure connections
jppf.management.ssl.port	44493	Deprecated: <i>the secure management port is now the same as the server port given in "jppf.ssl.server.port"</i> Secure JMX server port
jppf.node.idle	true	Whether a node is idle. This property is only set within a server.
jppf.node.management.port	11198	Node management port (to distinguish from server management port when local node is on)
jppf.node.max.jobs	Integer.MAX_VALUE	Maximum number of jobs that can be handled concurrently by a node
jppf.peer.<peer_name>.pool.size	1	Connection pool size for a manually configured peer driver connection - <i>peer_name</i> : one of the names defined in 'jppf.peers'
jppf.peer.<peer_name>.recovery.enabled	false	Heartbeat enabled flag for a manually configured peer driver connection. - <i>peer_name</i> : one of the names defined in 'jppf.peers'
jppf.peer.<peer_name>.server.host	localhost	Server host for a manually configured peer driver connection - <i>peer_name</i> : one of the names defined in 'jppf.peers'
jppf.peer.<peer_name>.server.port	11111	Server port for a manually configured peer driver connection - <i>peer_name</i> : one of the names defined in 'jppf.peers'
jppf.peer.<peer_name>.ssl.enabled	false	SSL enabled flag for a manually configured peer driver connection - <i>peer_name</i> : one of the names defined in 'jppf.peers'
jppf.peer.allow.orphans	false	Whether to send jobs to orphan peer servers
jppf.peer.discovery.enabled	false	Enable/disable peer server discovery
jppf.peer.pool.size	1	Size of discovered peer server connection pools

Name	Default value	Description
jppf.peer.recovery.enabled	false	Heartbeat enabled flag for a discovered peer driver connection.
jppf.peer.ssl.enabled	false	Toggle secure connections to remote peer servers
jppf.peers	null	Space-separated list of peer server names
jppf.peers.load.balance.threshold	Integer.MAX_VALUE	The number of connected nodes below which this driver load-balances to other peer drivers
jppf.recovery.enabled	false	Enable/disable recovery from hardware failures through a heartbeat mechanism
jppf.recovery.max.retries	3	Maximum number of pings to the node before the connection is considered broken
jppf.recovery.read.timeout	15000	Maximum ping response time from the node
jppf.recovery.reaper.pool.size	available processors	Number of threads allocated to the node connection reaper
jppf.redirect.err	null	File to redirect System.err to
jppf.redirect.err.append	false	Append to existing file (true) or create a new one (false)
jppf.redirect.out	null	File to redirect System.out to
jppf.redirect.out.append	false	Append to existing file (true) or create a new one (false)
jppf.resolve.addresses	true	Whether to resolve IP addresses
jppf.server.exitOnShutdown	false	Whether to exit the JVM when shutting the driver down
jppf.server.port	11111	Server port
jppf.ssl.client.distinct.truststore	false	Whether to 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.configuration.file	null	Path to the SSL configuration in the file system or classpath
jppf.ssl.configuration.source	null	SSL configuration as an arbitrary source
jppf.ssl.enabled	false	Enabled/disable secure connections
jppf.ssl.server.port	-1	Server port number for secure connections

12.2 Node properties

Name	Default value	Description
jppf.classloader.cache.size	50	Size of the class loader cache for the node
jppf.classloader.delegation	parent	Class loader delegation mode: 'parent' or 'url'
jppf.classloader.file.lookup	true	Enable/disable lookup of classpath resources in the file system
jppf.config.overrides.path	config/config-overrides.properties	Path to the temporary config overrides properties file
jppf.discovery.enabled	true	Enable/disable server discovery via UDP multicast
jppf.discovery.exclude.ipv4	null	IPv4 exclusion patterns for server discovery
jppf.discovery.exclude.ipv6	null	IPv6 exclusion patterns for server discovery
jppf.discovery.group	230.0.0.1	Server discovery: UDP multicast group
jppf.discovery.include.ipv4	null	IPv4 inclusion patterns for server discovery
jppf.discovery.include.ipv6	null	IPv6 inclusion patterns for server discovery
jppf.discovery.port	11111	Server discovery: UDP multicast port
jppf.discovery.timeout	1000	Server discovery timeout in milliseconds
jppf.idle.interruptIfRunning	true	Node idle mode: whether to shutdown the node at once when user activity resumes or wait until the node is no longer executing tasks
jppf.idle.mode.enabled	false	Enable/disable the idle mode
jppf.idle.poll.interval	1000	Node idle mode: how often the node will check for keyboard and mouse inactivity
jppf.idle.timeout	300000	Node idle mode: the time of keyboard and mouse inactivity before considering the node idle
jppf.java.path	null	Full path to the Java executable
jppf.jvm.options	null	JVM options for the node or server process

Name	Default value	Description
jppf.management.enabled	true	Enable/disable management of the node or server
jppf.management.host	null	Management server host
jppf.management.port	11198	Management remote connector port
jppf.management.server.forwarder	null	Fully qualified class name of a MBeanServerForwarder implementation with optional space-separated string parameters
jppf.node.android	false	Whether the node is an Android node
jppf.node.classloading.batch.period	100	How often batched class loading requests are sent to the server
jppf.node.management.port	11198	Node management port (to distinguish from server management port when local node is on)
jppf.node.max.jobs	Integer.MAX_VALUE	Maximum number of jobs that can be handled concurrently by a node
jppf.node.offline	false	Whether the node runs in offline mode
jppf.node.provisioning.master	true	Whether the node is a master node
jppf.node.provisioning.master.uuid	null	UUID of the master node for a given slave node
jppf.node.provisioning.slave	false	Whether the node is a slave node
jppf.node.provisioning.slave.config.path	config	Directory where slave-specific configuration files are located
jppf.node.provisioning.slave.id	-1	Id of a slave node generated by/scoped by its master. Unique within a single master's scope. Associate with "jppf.node.provisioning.master.uuid" to provide a globally unique id.
jppf.node.provisioning.slave.jvm.options	null	JVM options always added to the slave startup command
jppf.node.provisioning.slave.path.prefix	slave_nodes/ node_	Path prefix for the root directory of slave nodes
jppf.node.provisioning.startup.overrides.file	null	Path to an optional config overrides file for slaves launched at startup
jppf.node.provisioning.startup.overrides.source	null	An optional config overrides source (name of a class implementing org.jppf.utils.JPPFConfiguration.ConfigurationSourceReader or org.jppf.utils.JPPFConfiguration.ConfigurationSource) for slaves launched at startup
jppf.node.provisioning.startup.slaves	0	Number of slaves to launch upon master node startup
jppf.node.throttling.check.period	2000	How often the node throttling mechanism will check, expressed as an interval in milliseconds
jppf.processing.threads	available processors	Number of processing threads in the node
jppf.recovery.enabled	false	Enable/disable recovery from hardware failures through a heartbeat mechanism
jppf.redirect.err	null	File to redirect System.err to
jppf.redirect.err.append	false	Append to existing file (true) or create a new one (false)
jppf.redirect.out	null	File to redirect System.out to
jppf.redirect.out.append	false	Append to existing file (true) or create a new one (false)
jppf.resolve.addresses	true	Whether to resolve IP addresses
jppf.resource.cache.dir	sys.property "java.io.tmpdir"	Root location of the file-persisted caches
jppf.resource.cache.enabled	true	Whether the class loader resource cache is enabled
jppf.resource.cache.storage	file	Type of cache storage: either 'file' or 'memory'
jppf.server.connection.strategy	null	Fully qualified name of a class implementing org.jppf.node.connection.DriverConnectionStrategy
jppf.server.host	localhost	Server host name or IP address
jppf.server.port	11111	Server port
jppf.ssl.configuration.file	null	Path to the SSL configuration in the file system or classpath
jppf.ssl.configuration.source	null	SSL configuration as an arbitrary source
jppf.ssl.enabled	false	Enabled/disable secure connections
jppf.thread.manager.class	default	Type of thread pool to use in the node: either 'default' or 'org.jppf.server.node.fj.ThreadManagerForkJoin'

12.3 Node screensaver properties

Name	Default value	Description
jppf.screensaver.centerimage	org/jppf/node/jppf@home.gif	Path to the larger image at the center of the screen (built-in default screensaver)
jppf.screensaver.class	null	Class name of an implementation of org.jppf.node.screensaver.JPPFScreenSaver
jppf.screensaver.enabled	false	Enable/disable the screen saver
jppf.screensaver.fullscreen	false	Whether to display the screen saver in full screen mode
jppf.screensaver.handle.collisions	true	Handle collisions between moving logos (built-in default screensaver)
jppf.screensaver.height	800	Height in pixels (windowed mode)
jppf.screensaver.icon	org/jppf/node/jppf-icon.gif	Path to the image for the frame's icon (windowed mode)
jppf.screensaver.location.x	0	Screensaver's on-screen X coordinate (windowed mode)
jppf.screensaver.location.y	0	Screensaver's on-screen Y coordinate (windowed mode)
jppf.screensaver.logo.path	org/jppf/node/jppf_group_small.gif	Path(s) to the moving logo image(s) (built-in default screensaver)
jppf.screensaver.logos	10	Number of moving moving logos (built-in default screensaver)
jppf.screensaver.mouse.motion.close	true	Whether to close the screensaver on mouse motion (full screen mode)
jppf.screensaver.mouse.motion.delay	500	internal use
jppf.screensaver.node.listener	null	Class name of an implementation of org.jppf.node.screensaver.NodeIntegration
jppf.screensaver.speed	100	Speed of moving moving logos! from 1 to 100 (built-in default screensaver)
jppf.screensaver.status.panel.alignment	center	Horizontal alignment of the status panel (built-in default screensaver)
jppf.screensaver.title	JPPF screensaver	Title of the screensaver's JFrame in windowed mode
jppf.screensaver.width	1000	Width in pixels (windowed mode)

12.4 Client properties

Name	Default value	Description
<driver_name>.jppf.jmx.pool.size	1	Manually defined JMX connection pool size for a client-to-driver connection - <i>driver_name</i> : one of the driver names in 'jppf.drivers'
<driver_name>.jppf.max.jobs	Integer.MAX_VALUE	Manually defined maximum number of jobs that can be handled concurrently by a single connection - <i>driver_name</i> : one of the driver names in 'jppf.drivers'
<driver_name>.jppf.pool.size	1	Manually defined connection pool size for a client-to-driver connection - <i>driver_name</i> : one of the driver names in 'jppf.drivers'
<driver_name>.jppf.priority	0	Manually defined priority for a client-to-driver connection - <i>driver_name</i> : one of the driver names in 'jppf.drivers'
<driver_name>.jppf.recovery.enabled	false	Heartbeat enabled flag for a manually configured peer client connection - <i>driver_name</i> : one of the driver names in 'jppf.drivers'
<driver_name>.jppf.server.host	localhost	Manually defined driver host for a client-to-driver connection - <i>driver_name</i> : one of the driver names in 'jppf.drivers'
<driver_name>.jppf.server.port	11111	Manually defined driver port for a client-to-driver connection - <i>driver_name</i> : one of the driver names in 'jppf.drivers'
<driver_name>.jppf.ssl.enabled	false	Manually defined SSL enabled flag for a client-to-driver connection - <i>driver_name</i> : one of the driver names in 'jppf.drivers'
jppf.discovery.acceptMultipleInterfaces	false	Whether to discover server connections from multiple network interfaces
jppf.discovery.enabled	true	Enable/disable server discovery via UDP multicast

Name	Default value	Description
jppf.discovery.exclude.ipv4	null	IPv4 exclusion patterns for server discovery
jppf.discovery.exclude.ipv6	null	IPv6 exclusion patterns for server discovery
jppf.discovery.group	230.0.0.1	Server discovery: UDP multicast group
jppf.discovery.include.ipv4	null	IPv4 inclusion patterns for server discovery
jppf.discovery.include.ipv6	null	IPv6 inclusion patterns for server discovery
jppf.discovery.port	11111	Server discovery: UDP multicast port
jppf.discovery.priority	0	Priority assigned to discovered server connections (client/admin console)
jppf.discovery.timeout	1000	Server discovery timeout in milliseconds
jppf.drivers	default-driver	Names of the manually configured servers in the client
jppf.jmx.pool.size	1	JMX connection pool size when discovery is enabled
jppf.job.client.sla.default.policy	null	A default client-side execution policy to associate with submitted jobs when they don't have one
jppf.job.sla.default.policy	null	A default driver-side execution policy to associate with submitted jobs when they don't have one
jppf.load.balancing.algorithm	proportional	Load balancing algorithm name
jppf.load.balancing.persistence	null	Class name of the implementation of a load-balancer persistence in the driver or client, with optional parameters
jppf.load.balancing.persistence.hash	SHA-1	The hash function used to generate load-balancer state identifiers
jppf.load.balancing.profile	jppf	Load balancing parameters profile name
jppf.local.execution.enabled	false	Enable/disable local execution in the client
jppf.local.execution.priority	0	Priority assigned to the client local executor
jppf.local.execution.threads	available processors	Maximum threads to use for local execution
jppf.max.jobs	Integer.MAX_VALUE	Maximum number of jobs that can be handled concurrently by a single connection for server connections discovered via UDP multicast
jppf.pool.size	1	Connection pool size for server connections discovered via UDP multicast
jppf.recovery.enabled	false	Enable/disable recovery from hardware failures through a heartbeat mechanism
jppf.remote.execution.enabled	true	Enable/disable remote execution (client only)
jppf.resolve.addresses	true	Whether to resolve IP addresses
jppf.server.host	localhost	Server host name or IP address
jppf.server.port	11111	Server port
jppf.ssl.configuration.file	null	Path to the SSL configuration in the file system or classpath
jppf.ssl.configuration.source	null	SSL configuration as an arbitrary source
jppf.ssl.enabled	false	Enabled/disable secure connections

12.5 Desktop console properties

Name	Default value	Description
jppf.admin.console.view.<view_name>.addto	Main	The built-in view a pluggable view is attached to. It must be one of the tabbed panes of the admin console. Possible values: Main Topology Charts - <i>view_name</i> : a user-assigned name for the view
jppf.admin.console.view.<view_name>.autoselect	false	Whether to automatically select the pluggable view - <i>view_name</i> : a user-assigned name for the view
jppf.admin.console.view.<view_name>.class	null	Name of a pluggable view class, extending org.jppf.ui.plugin.PluggableView - <i>view_name</i> : a user-assigned name for the view
jppf.admin.console.view.<view_name>.enabled	true	Enable / disable a pluggable view - <i>view_name</i> : a user-assigned name for the view
jppf.admin.console.view.<view_name>.icon	null	Path to the icon for a pluggable view, seen as the tab icon - <i>view_name</i> : a user-assigned name for the view

Name	Default value	Description
jppf.admin.console.view.<view_name>.position	-1	The position at which a pluggable view is inserted within the enclosing tabbed pane. A negative value means insert at the end - <i>view_name</i> : a user-assigned name for the view
jppf.admin.console.view.<view_name>.title	null	The title for the view, seen as the tab label - <i>view_name</i> : a user-assigned name for the view
jppf.ui.default.scrollbar.thickness	10	The default thickness of the scrollbars in the GUI
jppf.ui.splash	true	Whether to display the animated splash screen at console startup, defaults to false
jppf.ui.splash.delay	500	Interval between images in milliseconds
jppf.ui.splash.images	null	One or more paths to the images displayed in a rolling sequence (like a slide show), separated by ' ' (pipe) characters
jppf.ui.splash.message	empty string	The fixed text displayed at center of the splash screen
jppf.ui.splash.message.color	64, 64, 128	The color of the fixed text displayed at center of the splash screen, as an 'r, g, b' or 'r, g, b, a' value

12.6 Web console properties

Name	Default value	Description
jppf.web.admin.refresh.interval	3	Interval in seconds between 2 refreshes of a page in the web admin console

12.7 7 Desktop and Web consoles properties

Name	Default value	Description
jppf.admin.refresh.interval.health	3000	Interval between updates of the JVM health data
jppf.admin.refresh.interval.stats	1000	Interval between updates of the server statistics
jppf.admin.refresh.interval.topology	1000	Interval between updates of the topology views
jppf.admin.refresh.system.info	false	Whether to refresh the nodes' system info as well (to use for node filtering on the client side)
jppf.gui.publish.mode	immediate_notifications	UI refresh mode for the job data panel: 'immediate_notifications' 'deferred_notifications' 'polling'
jppf.gui.publish.period	1000	Interval between updates of the job data view

12.8 Common properties

Name	Default value	Description
jppf.check.low.memory	true	Whether to check for low memory and trigger disk offloading
jppf.classloader.file.lookup	true	Enable/disable lookup of classpath resources in the file system
jppf.disk.overflow.threshold	2.0	Ratio of available heap over the size of an object to deserialize, below which disk overflow is triggered
jppf.gc.on.disk.overflow	true	Whether to call System.gc() and recompute the available heap size before triggering disk overflow
jppf.jmxremote.protocol	jppf	The JMX remote protocol
jppf.length.buffer.pool.size	100	Temporary buffer pool size for reading lengths as ints (size 4)
jppf.low.memory.threshold	32	Minimum heap size in MB below which disk overflow is systematically triggered, to avoid heap fragmentation and ensure there's enough memory to deserialize job headers
jppf.notification.offload.memory.threshold	80% of max heap size	Used heap in bytes above which notifications from task are offloaded to file. Defaults to 0.8 * maxHeapSize.
jppf.object.serialization.class	null	Serialization scheme: name of a class implementing org.jppf.serialization.JPPFSerialization
jppf.reconnect.initial.delay	0	Delay in seconds before the first (re)connection attempt
jppf.reconnect.interval	1	Frequency in seconds of reconnection attempts
jppf.reconnect.max.time	60	Time in seconds after which reconnection attempts stop. A negative value means never stop
jppf.resource.cache.dir	sys.property "java.io.tmpdir"	Root location of the file-persisted caches

Name	Default value	Description
jppf.script.default.language	javascript	Default script language for scripted property values
jppf.socket.buffer.size	32768	Receive/send buffer size for socket connections
jppf.socket.keepalive	false	Enable/disable socket keepalive
jppf.socket.max-idle	-1	Seconds a socket connection can remain idle before being closed (client only)
jppf.socket.tcp_nodelay	true	Enable/disable Nagle's algorithm
jppf.temp.buffer.pool.size	10	Maximum size of temporary buffers pool
jppf.temp.buffer.size	32768	Size of temporary buffers used in I/O transfers

12.9 .Net properties

Name	Default value	Description
jppf.dotnet.bridge.initialized	false	Whether the node is .Net-enabled

12.10 SSL/TLS properties

Name	Default value	Description
jppf.ssl.cipher.suites	null	Space-separated enabled cipher suites
jppf.ssl.client.auth	none	SSL client authentication level: 'none' 'want' 'need'
jppf.ssl.client.distinct.truststore	false	Whether to 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.password	null	Plain text client trust store password
jppf.ssl.client.truststore.password.source	null	Client trust store password as an arbitrary source
jppf.ssl.client.truststore.source	null	Client trust store location as an arbitrary source
jppf.ssl.client.truststore.type	jks	Client trust store format, e.g. 'JKS'
jppf.ssl.context.protocol	TLSv1.2	javax.net.ssl.SSLContext protocol
jppf.ssl.keystore.file	null	Path to the key store in the file system or classpath
jppf.ssl.keystore.password	null	Plain text key store password
jppf.ssl.keystore.password.source	null	Key store password as an arbitrary source
jppf.ssl.keystore.type	jks	Key store format, e.g. 'JKS'
jppf.ssl.keystore.source	null	Key store location as an arbitrary source
jppf.ssl.protocols	null	A list of space-separated enabled protocols
jppf.ssl.truststore.file	null	Path to the trust store in the file system or classpath
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.truststore.source	null	Trust store location as an arbitrary source
jppf.ssl.truststore.type	jks	Trust store format, e.g. 'JKS'

12.11 Memory usage optimization properties

Name	Default value	Description
jppf.check.low.memory	true	Whether to check for low memory and trigger disk offloading
jppf.disk.overflow.threshold	2.0	Ratio of available heap over the size of an object to deserialize, below which disk overflow is triggered
jppf.gc.on.disk.overflow	true	Whether to call System.gc() and recompute the available heap size before triggering disk overflow
jppf.low.memory.threshold	32	Minimum heap size in MB below which disk overflow is systematically triggered, to avoid heap fragmentation and ensure there's enough memory to deserialize job headers
jppf.notification.offload.memory.threshold	80% of max heap size	Used heap in bytes above which notifications from task are offloaded to file. Defaults to 0.8 * maxHeapSize.

13 Execution policy reference

Note: the execution policy XML schema can be found at: <https://www.jppf.org/schemas/ExecutionPolicy.xsd>

13.1 Execution Policy Elements

13.1.1 NOT

Negates a test

Class name: [ExecutionPolicy.NotRule](#)

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>
```

13.1.2 AND

Combines multiple tests through a logical AND operator

Class name: [ExecutionPolicy.AndRule](#)

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>
```

13.1.3 OR

Combines multiple tests through a logical OR operator

Class name: [ExecutionPolicy.OrRule](#)

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>

```

13.1.4 XOR

Combines multiple tests through a logical XOR operator

Class name: [ExecutionPolicy.XorRule](#)

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>

```

13.1.5 Equal

Performs a test of type "a = b". The value can be either numeric, boolean or a string.

Class name: [Equal](#)

Constructors:

```

public Equal(ValueType valueType, String propertyNameOrExpression, String value)
Equal(String propertyNameOrExpression, boolean ignoreCase, String value)
Equal(String propertyNameOrExpression, double value)
Equal(String propertyNameOrExpression, boolean value)

```

Usage:

```

policy = new Equal(ValueType.NUMERIC, "jppf.processing.threads",
  "$${ 2 * ${availableProcessors} }$");
policy = new Equal("some.property", true, "some_value");
policy = new Equal("${script{ 2 * ${some.property} }$", 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 or expression of the related type, min = 1, max = 1
<Value> : literal value or expression of the related type to compare with, min = 1, max = 1

Usage:

```

<Equal valueType="numeric">
  <Property>jppf.processing.threads</Property>
  <Value>${script{ 2 * ${availableProcessors} }}$</Value>
</Equal>

```



```
<Equal ignoreCase="true" valueType="string">
  <Property>some.property</Property>
  <Value>some value here</Value>
</Equal>
```

```
<Equal valueType="numeric">
  <Property>${script{ 2 * ${some.property} }}</Property>
  <Value>15</Value>
</Equal>
```

```
<Equal valueType="boolean">
  <Property>some.property</Property>
  <Value>true</Value>
</Equal>
```

13.1.6 LessThan

Performs a test of type "a < b"
The values can only be numeric.

Class name: [LessThan](#)

Constructor:

```
LessThan(String propertyNameOrExpression, double value)
LessThan(String propertyNameOrExpression, String expression)
```

Usage:

```
policy = new LessThan("some.property", 15.50);
policy = new LessThan("some.property", "${script{ 10 + 5.5 }}");
```

XML Element: **<LessThan>**

Nested elements:

<Property> : name of a node property or numeric expression, min = 1, max = 1

<Value> : literal value or expression to compare with, min = 1, max = 1

Usage:

```
<LessThan>
  <Property>some.property</Property>
  <Value>15.50</Value>
</LessThan>
```

```
<LessThan>
  <Property>some.property</Property>
  <Value>${script{ 10 + 5.5 }}</Value>
</LessThan>
```

13.1.7 AtMost

Performs a test of type "a <= b"
The value can only be numeric.

Class name: [AtMost](#)

Constructor:

```
AtMost(String propertyNameOrExpression, double value)
AtMost(String propertyNameOrExpression, String expression)
```

Usage:

```
policy = new AtMost("some.property", 15.50);
policy = new AtMost("some.property", "${script{ 10 + 5.5 }}");
```

XML Element: **<AtMost>**

Nested elements:

<Property> : name of a node property or numeric expression, min = 1, max = 1

<Value> : literal value or expression to compare with, min = 1, max = 1

Usage:

```
<AtMost>
  <Property>some.property</Property>
  <Value>15.49</Value>
</AtMost>
```

```
<AtMost>
  <Property>some.property</Property>
  <Value>$script{ 10 + 5.5 }$</Value>
</AtMost>
```

13.1.8 MoreThan

Performs a test of type "a > b"
The value can only be numeric.

Class name: [MoreThan](#)

Constructor:

```
MoreThan(String propertyNameOrExpression, double value)
MoreThan(String propertyNameOrExpression, String expression)
```

Usage:

```
policy = new MoreThan("some.property", 15.50);
policy = new MoreThan("some.property", "$script{ 10 + 5.5 }$");
```

XML Element: **<MoreThan>**

Nested elements:

<Property> : name of a node property or numeric expression, min = 1, max = 1

<Value> : literal value or expression to compare with, min = 1, max = 1

Usage:

```
<MoreThan>
  <Property>some.property</Property>
  <Value>15.50</Value>
</MoreThan>
```

```
<MoreThan>
  <Property>some.property</Property>
  <Value>$script{ 10 + 5.5 }$</Value>
</MoreThan>
```

13.1.9 AtLeast

Performs a test of type "property_value >= value"
The value can only be numeric.

Class name: [AtLeast](#)

Constructor:

```
AtLeast(String propertyNameOrExpression, double value)
AtLeast(String propertyNameOrExpression, String expression)
```

Usage:

```
policy = new AtLeast("some.property", 15.51);
policy = new AtLeast("some.property", "$script{ 10 + 5.5 }$");
```

XML Element: **<AtLeast>**

Nested elements:

<Property> : name of a node property or numeric expression, min = 1, max = 1

<Value> : literal value or expression to compare with, min = 1, max = 1

Usage:

```
<AtLeast>
  <Property>some.property</Property>
  <Value>15.51</Value>
</AtLeast>
```

```
<AtLeast>
  <Property>some.property</Property>
  <Value>$script{ 10 + 5.5 }$</Value>
</AtLeast>
```

13.1.10 BetweenII

Performs a test of type "a <= value <= b" (range interval with lower and upper bounds included)
The values a and b can only be numeric.

Class name: [BetweenII](#)

Constructors:

```
BetweenII(String propertyNameOrExpression, double a, double b)
BetweenII(String propertyNameOrExpression, String a, double b)
BetweenII(String propertyNameOrExpression, double a, String b)
BetweenII(String propertyNameOrExpression, String a, String b)
```

Usage:

```
policy = new BetweenII("my.prop", 1, 3);
policy = new BetweenII("$script{2 * ${other.prop}}$", "$script{1 + 0.5}$", 3);
policy = new BetweenII("my.prop", 1, "$script{2 * ${other.prop}}$");
policy = new BetweenII("my.prop", "$script{2 * ${prop1}}$", "${prop2}");
```

XML Element: **<BetweenII>**

Nested elements:

<Property> : name of a node property or numeric expression, min = 1, max = 1

<Value> : numeric literals or expressions for the bounds of the interval, min = 2, max = 2

Usage:

```
<BetweenII>
  <Property>my.prop</Property>
  <Value>1</Value>
  <Value>3</Value>
</BetweenII>
```

```
<BetweenII>
  <Property>$script{2 * ${other.prop}}$</Property>
  <Value>$script{1 + 0.5}$</Value>
  <Value>3</Value>
</BetweenII>
```

```
<BetweenII>
  <Property>my.prop</Property>
  <Value>1</Value>
  <Value>$script{2 * ${other.prop}}$</Value>
</BetweenII>
```

```
<BetweenII>
  <Property>my.prop</Property>
  <Value>$script{2 * ${prop1}}$</Value>
  <Value>${prop2}</Value>
</BetweenII>
```

13.1.11 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: [BetweenIE](#)

Constructor:

```
BetweenIE(String propertyNameOrExpression, double a, double b)
BetweenIE(String propertyNameOrExpression, String a, double b)
BetweenIE(String propertyNameOrExpression, double a, String b)
BetweenIE(String propertyNameOrExpression, String a, String b)
```

Usage:

```
policy = new BetweenIE("my.prop", 1, 3);
policy = new BetweenIE("$script{2 * ${other.prop}}$", "$script{1 + 0.5}$", 3);
policy = new BetweenIE("my.prop", 1, "$script{2 * ${other.prop}}$");
policy = new BetweenIE("my.prop", "$script{2 * ${prop1}}$", "${prop2}");
```

XML Element: **<BetweenIE>**

Nested elements:

<Property> : name of a node property or numeric expression, min = 1, max = 1

<Value> : numeric literals or expressions for the bounds of the interval, min = 2, max = 2

Usage:

```
<BetweenIE>
  <Property>my.prop</Property>
  <Value>1</Value>
  <Value>3</Value>
</BetweenIE>
```

```
<BetweenIE>
  <Property>$script{2 * ${other.prop}}$</Property>
  <Value>$script{1 + 0.5}$</Value>
  <Value>3</Value>
```

```
</BetweenIE>
```

```
<BetweenIE>
  <Property>my.prop</Property>
  <Value>1</Value>
  <Value>${script{2 * ${other.prop}}}$</Value>
</BetweenIE>
```

```
<BetweenIE>
  <Property>my.prop</Property>
  <Value>${script{2 * ${prop1}}}$</Value>
  <Value>${prop2}</Value>
</BetweenIE>
```

13.1.12 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: [BetweenEI](#)

Constructor:

```
BetweenEI(String propertyNameOrExpression, double a, double b)
BetweenEI(String propertyNameOrExpression, String a, double b)
BetweenEI(String propertyNameOrExpression, double a, String b)
BetweenEI(String propertyNameOrExpression, String a, String b)
```

Usage:

```
policy = new BetweenEI("my.prop", 1, 3);
policy = new BetweenEI("${script{2 * ${other.prop}}}$", "${script{1 + 0.5}}$", 3);
policy = new BetweenEI("my.prop", 1, "${script{2 * ${other.prop}}}$");
policy = new BetweenEI("my.prop", "${script{2 * ${prop1}}}$", "${prop2}");
```

XML Element: **<BetweenEI>**

Nested elements:

<Property> : name of a node property or numeric expression, min = 1, max = 1

<Value> : numeric literals or expressions for the bounds of the interval, min = 2, max = 2

Usage:

```
<BetweenEI>
  <Property>my.prop</Property>
  <Value>1</Value>
  <Value>3</Value>
</BetweenEI>
```

```
<BetweenEI>
  <Property>${script{2 * ${other.prop}}}$</Property>
  <Value>${script{1 + 0.5}}$</Value>
  <Value>3</Value>
</BetweenEI>
```

```
<BetweenEI>
  <Property>my.prop</Property>
  <Value>1</Value>
  <Value>${script{2 * ${other.prop}}}$</Value>
</BetweenEI>
```

```
<BetweenEI>
  <Property>my.prop</Property>
  <Value>${script{2 * ${prop1}}}$</Value>
  <Value>${prop2}</Value>
</BetweenEI>
```

13.1.13 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: [BetweenEE](#)

Constructor:

```
BetweenEE(String propertyNameOrExpression, double a, double b)
BetweenEE(String propertyNameOrExpression, String a, double b)
BetweenEE(String propertyNameOrExpression, double a, String b)
BetweenEE(String propertyNameOrExpression, String a, String b)
```

Usage:

```
policy = new BetweenEE("my.prop", 1, 3);
```

```
policy = new BetweenEE("$script{2 * ${other.prop}}$", "$script{1 + 0.5}$", 3);
policy = new BetweenEE("my.prop", 1, "$script{2 * ${other.prop}}$");
policy = new BetweenEE("my.prop", "$script{2 * ${prop1}}$", "${prop2}");
```

XML Element: **<BetweenEE>**

Nested elements:

<Property> : name of a node property or numeric expression, min = 1, max = 1

<Value> : numeric literals or expressions for the bounds of the interval, min = 2, max = 2

Usage:

```
<BetweenEE>
  <Property>my.prop</Property>
  <Value>1</Value>
  <Value>3</Value>
</BetweenEE>
```

```
<BetweenEE>
  <Property>$script{2 * ${other.prop}}$</Property>
  <Value>$script{1 + 0.5}$</Value>
  <Value>3</Value>
</BetweenEE>
```

```
<BetweenEE>
  <Property>my.prop</Property>
  <Value>1</Value>
  <Value>$script{2 * ${other.prop}}$</Value>
</BetweenEE>
```

```
<BetweenEE>
  <Property>my.prop</Property>
  <Value>$script{2 * ${prop1}}$</Value>
  <Value>${prop2}</Value>
</BetweenEE>
```

13.1.14 Contains

Performs a test of type “property_value contains substring”

The value can be only a string.

Class name: [Contains](#)

Constructor:

```
Contains(String propertyNameOrExpression, boolean ignoreCase, String valueOrExpression)
```

Usage:

```
policy = new Contains("some.property", true, "$script{'${other.prop}' + 'abc'}$");
```

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 literal or expression to lookup, min = 1, max = 1

Usage:

```
<Contains ignoreCase="true">
  <Property>some.property</Property>
  <Value>$script{'${other.prop}' + 'abc'}$</Value>
</Contains>
```

13.1.15 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: [OneOf](#)

Constructors:

```
OneOf(String propertyNameOrExpression, boolean ignoreCase, String...valuesOrExpressions)
```

```
OneOf(String propertyNameOrExpression, double...values)
```

```
OneOf(String propertyNameOrExpression, String...numericValuesOrExpressions)
```

Usage:

```
policy = new OneOf("user.language", true, "en", "fr", "it");
policy = new OneOf("$script{${prop1} + 2.1}$", 1.2, 5.1);
policy = new OneOf("my.prop", "1.2", "$script{1.2}$", "$script{${other.prop} + 2.1}$");
```

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 or expression of the related type, min = 1, max = 1

<Value> : literal or expression of the related type, min = 1, max = unbounded

Usage:

```
<OneOf ignoreCase="true">
  <Property>user.language</Property>
  <Value>en</Value>
  <Value>fr</Value>
  <Value>it</Value>
</OneOf>
```

```
<OneOf>
  <Property>$script{${prop1} + 2.1}$</Property>
  <Value>1.2</Value>
  <Value>5.1</Value>
</OneOf>
```

```
<OneOf>
  <Property>my.prop</Property>
  <Value>1.2</Value>
  <Value>$script{1.2}$</Value>
  <Value>$script{${other.prop} + 2.1}$</Value>
</OneOf>
```

13.1.16 RegExp

Performs a test of type "property_value_or_expression matches regular_expression"

The regular expression must follow the syntax for the [Java regular expression patterns](#).

Class name: **RegExp**

Constructor:

RegExp(String propertyNameOrExpression, String pattern)

Usage:

```
policy = new RegExp("some.property", "a.*z");
```

XML Element: **<RegExp>**

Nested elements:

<Property> : name of a node property or string expression, 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>
```

13.1.17 ScriptedPolicy

Executes a script which returns a boolean value.

Class name: **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>
```

13.1.18 CustomRule

Performs a user-defined test that can be specified in an XML policy document.

Class name: subclass of [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>
```

13.1.19 Preference

Evaluates a set of nested policies ordered by preference.

Class name: [Preference](#)

Constructors:

```
Preference(ExecutionPolicy...policies)
```

```
Preference(List<ExecutionPolicy> policies)
```

Usage:

```
policy = new Preference(AtLeast("jppf.processing.threads", 4),
  new LessThan("jppf.processing.threads", 4).and(new AtLeast("maxMemory", 1_000_000)));
```

XML Element: **<Preference>**

Usage:

```
<Preference>
  <AtLeast>
    <Property>jppf.processing.thread</Property>
    <Value>4</Value>
  </AtLeast>
  <AND>
    <LessThan>
      <Property>jppf.processing.thread</Property>
      <Value>4</Value>
    </LessThan>
    <AtLeast>
      <Property>maxMemory</Property>
      <Value>1000000</Value>
    </AtLeast>
  </AND>
</Preference>
```

13.1.20 IsInIPv4Subnet

Performs a test of type “[ipv4.addresses](#) has an address in at least one of s_1 , ... or s_n subnets”

Each subnet can be expressed in either CIDR or [IPv4AddressPattern](#) format.

Class name: [IsInIPv4Subnet](#)

Constructors:

```
IsInIPv4Subnet(String...subnets)
```

```
IsInIPv4Subnet(Collection<String> subnets)
```

Usage:

```
policy = new IsInIPv4Subnet("192.168.1.0/24", "192.168.1.0-255", "${ip4.netmask}");
```

XML element: **<IsInIPv4Subnet>**

Nested element: **<Subnet>** : IPv4 subnet mask, min = 1, max = unbounded

Usage:

```
<IsInIPv4Subnet>
  <Subnet>192.168.1.0/24</Subnet>
  <Subnet>192.168.1.0-255</Subnet>
  <Subnet>${ip4.netmask}</Subnet>
</IsInIPv4Subnet>
```


13.1.21 IsInIPv6Subnet

Performs a test of type “[ipv6.addresses](#) has an address in at least one of s_1, \dots or s_n subnets”
Each subnet can be expressed in either CIDR or [IPv6AddressPattern](#) format.

Class name: [IsInIPv6Subnet](#)

Constructors:

```
IsInIPv6Subnet(String...subnets)
IsInIPv6Subnet(Collection<String> subnets)
```

Usage:

```
policy = new IsInIPv6Subnet("::1/80", "1080::0:0:8:800:200C:417A/97", "${ip6.netmask}");
```

XML element: **<IsInIPv6Subnet>**

Nested element: **<Subnet>** : IPv6 subnet mask, min = 1, max = unbounded

Usage:

```
<IsInIPv6Subnet>
  <Subnet>::1/80</Subnet>
  <Subnet>1080::0:0:8:800:200C:417A/97</Subnet>
  <Subnet>${ip6.netmask}</Subnet>
</IsInIPv6Subnet>
```

13.1.22 NodesMatching

Counts the nodes matching a given execution policy and compares the resulting number with an expected number of nodes, using a specified comparison operator: *expected_number compared_to (actual_number matching node_policy)*.

Class name: [NodesMatching](#)

Constructors:

```
NodesMatching(Operator operator, long expectedNodes, ExecutionPolicy nodePolicy)
NodesMatching(Operator operator, long expectNodesExpression, ExecutionPolicy nodePolicy)
```

Usage:

```
new NodesMatching(Operator.AT_LEAST, 4, new MoreThan("availableProcessors", 2));
new NodesMatching(Operator.AT_LEAST, "$S${idle.nodes} / 2$",
  new MoreThan("availableProcessors", 2));
```

reads as: *at least 4 nodes with more than 2 processors each*

XML element: **<NodesMatching>**

Attributes:

- **operator**: the comparison operator, one of EQUAL, NOT_EQUAL, AT_LEAST, MORE_THAN, AT_MOST, LESS_THAN.
- **expected**: the number of expected nodes as an int literal or expression

Nested element: any other execution policy element

Usage:

```
<NodesMatching operator="AT_MOST" expected="4">
  <MoreThan>
    <Property>availableProcessors</Property>
    <Value>2</Value>
  </MoreThan>
</NodesMatching>
```

```
<NodesMatching operator="AT_MOST" expected="$S${idle.nodes} / 2$">
  <MoreThan>
    <Property>availableProcessors</Property>
    <Value>2</Value>
  </MoreThan>
</NodesMatching>
```

13.1.23 AcceptAll

An execution policy which accepts everything. Mostly useful when used within a more complex policy.

Class name: [AcceptAll](#)

Constructors:

```
AcceptAll()
AcceptAll(ExecutionPolicy wrappedPolicy)
```

Usage:

```
ExecutionPolicy policy = new AcceptAll();
policy = new AcceptAll(new AtLeast("availableProcessors", 2));
```

reads as: *accept whether the node has at least 2 CPUs or not*

XML element: **<AcceptAll>**

Optional nested element: any other execution policy element

Usage:

```
<AcceptAll/>
```

```
<AcceptAll>
  <AtLeast>
    <Property>availableProcessors</Property>
    <Value>2</Value>
  </AtLeast>
</AcceptAll>
```

13.1.24 RejectAll

An execution policy which rejects everything. Mostly useful when used within a more complex policy.

Class name: [RejectAll](#)

Constructors:

```
RejectAll()
```

```
RejectAll(ExecutionPolicy wrappedPolicy)
```

Usage:

```
ExecutionPolicy policy = new RejectAll();
policy = new RejectAll(new AtLeast("availableProcessors", 2));
```

reads as: reject *whether the node has at least 2 CPUs or not*

XML element: **<RejectAll>**

Optional nested element: any other execution policy element

Usage:

```
<RejectAll/>
```

```
<RejectAll>
  <AtLeast>
    <Property>availableProcessors</Property>
    <Value>2</Value>
  </AtLeast>
</RejectAll>
```

13.1.25 IsMasterNode

An execution policy which determines whether a node is a [master node](#).

Class name: [IsMasterNode](#)

Constructors:

```
IsMasterNode()
```

Usage:

```
ExecutionPolicy policy = new IsMasterNode();
```

XML element: **<IsMasterNode>**

No nested element.

Usage:

```
<IsMasterNode/>
```

```
<IsMasterNode></IsMasterNode>
```

13.1.26 IsSlaveNode

An execution policy which determines whether a node is a [slave node](#).

Class name: [IsSlaveNode](#)

Constructors:

```
IsSlaveNode()
```

Usage:

```
ExecutionPolicy policy = new IsSlaveNode();
```

XML element: **<IsSlaveNode>**

No nested element.

Usage:

```
<IsSlaveNode/>
```

```
<IsSlaveNode></IsSlaveNode>
```

13.1.27 IsLocalChannel

An execution policy which determines whether a connection endpoint is a local client executor (on the client side), or a local node (on the server side).

Class name: [**IsLocalChannel**](#)

Constructors:

```
IsLocalChannel()
```

Usage:

```
ExecutionPolicy policy = new IsLocalChannel();
```

XML element: **<IsLocalChannel>**

No nested element.

Usage:

```
<IsLocalChannel/>
```

```
<IsLocalChannel></IsLocalChannel>
```

13.1.28 IsPeerDriver

An execution policy which determines whether a node connection endpoint is actually a [remote peer driver](#).

Class name: [**IsPeerDriver**](#)

Constructors:

```
IsPeerDriver()
```

Usage:

```
ExecutionPolicy policy = new IsPeerDriver();
```

XML element: **<IsPeerDriver>**

No nested element.

Usage:

```
<IsPeerDriver/>
```

```
<IsPeerDriver></IsPeerDriver>
```

13.2 Execution policy properties

13.2.1 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
15. in `JPPFSystemInformation.getOS()` : operating system information properties

13.2.2 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\(\)](#)

13.2.3 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);
```

On the server side, the JPPF driver also dynamically sets and maintains a number of properties for each node:

- **jppf.peer.driver**: whether the node is actually a peer driver
- **jppf.peer.total.nodes**: the number of nodes connected to the remote peer driver
- **jppf.processing.threads**: for a node this is the number of threads, updated whenever it is changed via JMX. For a peer driver, this is the total number of processing threads for the nodes that are connected to it
- **jppf.node.idle**: whether the node is idle or busy executing tasks

Related APIs:

[JPPFSystemInformation.getJppf\(\)](#)
[JPPFConfiguration.getProperties\(\)](#)

13.2.4 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\(\)`](#)

13.2.5 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\(\)`](#)

13.2.6 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`](#)
[`java.lang.management.RuntimeMXBean`](#)

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
`usedMemory` : the used heap memory in bytes
`availableMemory` : the total available memory in bytes, equal to `maxMemory - usedMemory`
`startTime` : the JVM start time in milliseconds
`uptime` : the JVM uptime in milliseconds
`inputArgs` : the options passed to the JVM, not including the arguments passed to the `main()` method, formatted as a list of strings separated by ", " (comma followed by a space)

Note: *uptime, totalMemory and freeMemory are the values taken upon the node or server startup. They may have changed subsequently and should therefore only be used with appropriate precautions.*

13.2.7 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 `host_name|ipv4_address`
`ipv6.addresses` : space-separated list of IPV6 addresses with associated host in the format `host_name|ipv6_address`

Example:

```
ipv4.addresses = www.myhost.com|192.168.121.3 localhost|127.0.0.1 10.1.1.12|10.1.1.12
ipv6.addresses = www.myhost.com|2001:0db8:85a3:08d3:1319:8a2e:0370:7334
```

Note: when a host name cannot be resolved, the left-hand part of the address, on the left of the "|" (pipe character) will be set to the IP address

13.2.8 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
```

13.2.9 Operating system properties

Related APIs:

[JPPFSystemInformation.getOS\(\)](#)
[SystemUtils.getOS\(\)](#)

List of properties:

os.TotalPhysicalMemorySize: total system RAM in bytes

os.FreePhysicalMemorySize: available system RAM in bytes *

os.TotalSwapSpaceSize: total system swap space in bytes

os.FreeSwapSpaceSize: available system swap space in bytes *

os.CommittedVirtualMemorySize: process committed virtual memory in bytes *

os.ProcessCpuTime: process CPU time in nanoseconds *

os.Name: operating system name

os.Version: operating system version

os.Arch: operating system architecture

os.AvailableProcessors: system available cores

** these values are computed at node or server startup time, and will change subsequently; they should be used with appropriate precautions.*

Example:

```
os.Arch = amd64
os.AvailableProcessors = 8
os.CommittedVirtualMemorySize = 375021568
os.FreePhysicalMemorySize = 12983939072
os.FreeSwapSpaceSize = 29355577344
os.Name = Windows 7
os.ProcessCpuTime = 312002000
os.TotalPhysicalMemorySize = 17113022464
os.TotalSwapSpaceSize = 34224136192
os.Version = 6.1
```

13.2.10 Server statistics

Related APIs:

[JPPFSystemInformation.getStats\(\)](#)
[org.jppf.utils.stats.*](#)

Related documentation: see *Development guide > The JPPF statistics API*

The server statistics values defined as constants in JPPFStatistics are now included and available in the execution properties for a JPPF driver. The property names are formatted as "*statistic_label.attribute* = value" where:

- *statistic_label* is the string returned by [JPPFSnapshot.getLabel\(\)](#)
- *attribute* is the attribute name corresponding to one of the other getters in [JPPFSnapshot](#)
- value is always expressed as a double value, even if it is an integer

For a a cumulative or non-cumulative statistics snapshot, such as [JPPFStatisticHelper.IDLE_NODE](#), the available properties will thus be:

```
idle.nodes.avg = 1.0
idle.nodes.count = 1.0
idle.nodes.latest = 1.0
idle.nodes.max = 1.0
idle.nodes.min = 0.0
idle.nodes.total = 1.0
```

For a single value snapshot such as [JPPFStatisticHelper.CLIENT_IN_TRAFFIC](#), there is a single property:

```
client.traffic.in.total = 1589.0
```


14 JPPF Deployment

14.1 Drivers and nodes as services

In this section, it is assumed that you have downloaded and installed a JPPF driver distribution or one of the JPPF node distributions. The root installation folder of the driver or node will be designated as JPPF_HOME in the next sections.

14.1.1 Windows services with Apache's commons-daemon

Using the Apache commons-daemon utilities, it is possible to install a driver or node as a Windows service without any additional download. First a little bit of configuration checking and/or editing is required:

- a) edit the file **service.bat** in the **JPPF_HOME\bin\daemon** folder in your favorite text editor
- b) check the following environment variables at the start of the file and make sure they match the JVM root location and architecture in your environment:

```
rem must point to the root installation folder of a JDK or JRE
set JRE_HOME=%JAVA_HOME%
rem JVM architecture, must be one of 32, 64, ia64
set ARCH=64
```

If you need additional jar files or class folders for your node or driver, you can either drop them in the JPPF_HOME\lib folder or specify them with the ADDITIONAL_CP environment variable, for example:

```
rem add more jars and class folders to the default classpath
set ADDITIONAL_CP=C:/libs/myJar.jar;C:/MyProject/classes
```

When this is done, you are ready to install the JPPF driver or node as a Windows service:

```
service.bat install
```

To remove the service, use the following command:

```
service.bat uninstall
```

Lastly, the **gui.bat** script starts a utility which installs itself in the system tray and allows you to monitor, control and configure the service. Launch it with the following command:

```
gui.bat
```

14.1.2 Windows services with the Java Service Wrapper

JPPF drivers and nodes can be run as Windows Services using the Java Service Wrapper available at [Tanuki Software](#). This is done by following these steps:

- [download](#) the Java Service Wrapper for your platform and copy the files **wrapper.exe**, **wrapper.dll** and **wrapper.jar** to the JPPF_HOME directory
- edit the file **JPPF_HOME\config\wrapper-driver.conf** or **JPPF_HOME\config\wrapper-node.conf** and check that the setting for the **wrapper.java.command** property is valid (either the PATH environment must contain a JRE, or the JRE installation directory must be entered here)
- run the **InstallService.bat** script in the **JPPF_HOME\bin\jsw** folder to install the JPPF service
- run the **UninstallService.bat** script in the **JPPF_HOME\bin\jsw** folder to uninstall the JPPF service

14.1.3 Unix daemons with the Java Service Wrapper

JPPF drivers and nodes can be run as Windows Services using the Java Service Wrapper available at [Tanuki Software](#). This is done by following these steps:

- [download](#) the Java Service Wrapper for your platform and copy the files **wrapper**, **libwrapper.so** and **wrapper.jar** to the JPPF_HOME directory
- don't forget to set the executable bit for the JPPF_HOME/bin/jsw/JPPFDriver and the wrapper script/executable
- edit the file **JPPF_HOME/config/wrapper-driver.conf** or **JPPF_HOME/config/wrapper-node.conf** and check that the setting for the **wrapper.java.command** property is valid (either the PATH environment must contain a JRE, or the JRE installation directory must be entered here)
- open a terminal in the **JPPF_HOME/bin/jsw** directory
- to run the driver or node as a daemon: **./JPPFDriver start** or **./JPPFNode start**
- to stop the driver or node: **./JPPFDriver stop** or **./JPPFNode stop**
- to restart the driver or node: **./JPPFDriver restart** or **./JPPFNode restart**

14.2 Running JPPF on Amazon's EC2, Rackspace, or other Cloud Services

14.2.1 Java Cloud Toolkit

Apache jclouds® provides an open source Java toolkit for accessing EC2, Rackspace, and several other cloud providers. The Compute interface provides several methods for creating server(s) from either the provider's or your own saved image, file transfer, executing commands on the new server's shell, deleting servers, and more. Servers can be managed individually or in groups, permitting your JPPF client to provision and configure servers on-the-fly.

When you create a cloud server with this toolkit, you programmatically have access to its NodeMetadata including IP addresses and login credentials. By creating your driver and nodes in the right sequence, you can pass IP information between them, as well as create different server types and use Job SLA's based on the IPs to vary the types of servers you want for different types of jobs.

14.2.2 Server discovery

Cloud servers do 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, Rackspace, and others charge 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...

14.2.3 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.

Rackspace cloud servers have no default restrictions on private IPs and ports at the same datacenter, so JPPF will work out-of-the-box on an all-cloud network. If added security is desired, you can create an Isolated Cloud Network with your own set of private IP addresses (192.168.x.x). In order to associate cloud servers with a dedicated (managed) server at Rackspace, you must request to configure RackConnect to merge your cloud and managed accounts and use all-private IPs.

14.2.4 Instance type

EC2 and Rackspace nodes vary the number of available cores and available memory, so you may want a different node property file and startup script for each instance type you start, with an appropriate number of threads. For instance, on a EC2 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:

```
jppf.processing.threads = 9
```

If your tasks require more I/O, you may need to experiment to find the best completion rate. You may want to configure multiple JPPF nodes on the same server.

14.2.5 IP Addresses

All EC2 and Rackspace 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. The following approaches will probably work:

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 or Rackspace 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.

As of JPPF 4.1, you can use the `getPrivateAddresses()` method of the `jclouds NodeMetadata` class to return the private IP of the server, and then use the `runScriptOnNode()` method of the `ComputeService` class to set an environment variable or publish the IP in a file which can be referenced using substitutions or includes.

There are lots of other approaches that will give you the same results – just have the server publish its location to some known location (including possibly the node itself) and have the node read this and dynamically create its properties instead of having a fixed file.

14.3 Nodes in “Idle Host” mode

A node can be configured to run only when its host is considered idle, that is, when no keyboard or mouse activity has occurred for a specified time.

The idleness detection and behavior are specified with the following node configuration properties:

```
# enable/disable idle mode, defaults to false (disabled)
jppf.idle.mode.enabled = true

# Time of keyboard and mouse inactivity after which the system is considered
# idle, in milliseconds. Defaults to 300000 (5 minutes)
jppf.idle.timeout = 6000

# Interval between 2 successive calls to the native APIs to determine whether
# the system idle state has changed. Defaults to 1000 ms
jppf.idle.poll.interval = 1000

# Whether to shutdown the node immediately when a mouse/keyboard activity is detected,
# or wait until the node is no longer executing tasks. Default value is true (immediate
# shutdown)
jppf.idle.interruptIfRunning = true
```

With the above settings, the node will be started whenever there has been no user input for at least 6 seconds, the check for idleness will occur every second, and the node will stop immediately whenever the user resumes activity.

When the node is set to stop immediately upon user activity resumption, this will trigger the normal server behavior, which is to resubmit the tasks based on the resubmission settings at the job or task level.

Please note that the following node deployments are incompatible with the idle mode:

- local nodes running in the same JVM as a driver
- slave nodes, when using [node provisioning](#). However, the master node supports the idle mode

The idleness detection feature relies on a JNI bridge provided by the [JNA](#) libraries, provided in the node distribution. Consequently, it is not guaranteed to work with all platforms. It should work with all Windows versions from Windows Vista upwards, all Linux/BSD/Unix distributions which have the [XScreenSaver](#) library installed, and Mac OS/X. In particular, this feature may not be supported with some Unix-based OSes.

14.4 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.

14.4.1 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**.

14.4.2 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.

14.4.3 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);
```

14.5 Node provisioning

Any JPPF node has the ability to start new nodes on the same physical or virtual machine, and stop and monitor these nodes afterwards. This constitutes a node provisioning facility, which allows dynamically growing or shrinking a JPPF grid based on the workload requirements.

This provisioning ability establishes a master/slave relationship between a standard node (master) and the nodes that it starts (slaves). Please note that a slave node cannot be in turn used as a master. Apart from this restriction, slave nodes can be managed and monitored as any other node - unless they are defined as [offline nodes](#).

Please note that [offline nodes](#) cannot be used as master nodes, since they cannot be managed.

14.5.1 Provisioning with the JMX API

As seen in [Management and monitoring > Node management > Node provisioning](#), provisioning can be performed with an MBean implementing the [JPPFNodeProvisioningMBean](#) interface, defined as follows:

```
public interface JPPFNodeProvisioningMBean {
    // The object name of this MBean
    String MBEAN_NAME = "org.jppf:name=provisioning,type=node";
    // Get the number of slave nodes started by this MBean
    int getNbSlaves();
    // Start or stop the required number of slaves to reach the specified number
    void provisionSlaveNodes(int nbNodes);
    // Same action, explicitly specifying the interrupt flag
    void provisionSlaveNodes(int nbNodes, boolean interruptIfRunning);
    // Start or stop the required number of slaves to reach the specified number,
    // using the specified configuration overrides
    void provisionSlaveNodes(int nbNodes, TypedProperties configOverrides);
    // Same action, explicitly specifying the interrupt flag
    void provisionSlaveNodes(
        int nbNodes, boolean interruptIfRunning, TypedProperties configOverrides);
}
```

Combined with the ability to manage and monitor nodes via the server to which they are attached, this provides a powerful and sophisticated way to grow, shrink and control a JPPF grid on demand. Let's look at the following example, which shows how to provision new nodes with specific memory requirements, execute a job on these nodes, then restore the grid to its initial topology.

The first thing that we do is to initialize a JPPF client, obtain a JMX connection from the JPPF server, then get a reference to the server MBean which forwards management requests to the nodes:

```
JPPFClient client = new JPPFClient();
// wait until a standard connection to the driver is established
JPPFConnectionPool pool = client.awaitWorkingPool();
// wait until a JMX connection to the driver is established
JMXDriverConnectionWrapper jmxDriver =
    pool.awaitJMXConnections(Operator.AT_LEAST, 1, true).get(0);
// get a proxy to the mbean that forwards management requests to the nodes
JPPFNodeForwardingMBean forwarder = jmxDriver.getNodeForwarder();
```

In the next step, we will create a node selector, based on an execution policy which matches all master nodes:

```
// create a node selector which matches all master nodes
ExecutionPolicy masterPolicy = new Equal("jppf.node.provisioning.master", true);
NodeSelector masterSelector = new NodeSelector.ExecutionPolicySelector(masterPolicy);
```

Note the use of the configuration property "jppf.node.provisioning.master = true" which is present in every master node. Next, we define configuration overrides to fit our requirements:

```
TypedProperties overrides = new TypedProperties()
    // request 2 processing threads
    .setInt("jppf.processing.threads", 2)
    // specify a server JVM with 512 MB of heap
    .setString("jppf.jvm.options", "-server -Xmx512m");
```

Now we can provision the nodes we need:

```
// request that 2 slave nodes be provisioned, by invoking the provisionSlaveNodes()
// method on all nodes matched by the selector, with the configuration overrides
forwarder.provisionSlaveNodes (masterSelector, 2, overrides);

// give the nodes enough time to start the slaves
Thread.sleep(3000L);
```

We then check that our master nodes effectively have two slaves started:

```
// request the 'NbSlaves' Mbean attribute for each master
Map<String, Object> resultsMap = forwarder.getNbSlaves(masterSelector);

// keys in the map are node UUIDs
// the values are either integers if the request succeeded, or a Throwable if it failed
for (Map.Entry<String, Object> entry: resultsMap.entrySet()) {
    if (entry.getValue() instanceof Throwable) {
        System.out.println("node " + entry.getKey() + " raised " +
            ExceptionUtils.getStackTrace((Throwable) entry.getValue()));
    } else {
        System.out.println("master node " + entry.getKey() + " has " +
            entry.getValue() + " slaves");
    }
}
```

Once we are satisfied with the topology we just setup, we can submit a job on the slave nodes:

```
// create the job and add tasks
JPPFJob job = new JPPFJob();
job.setName("Hello World");
for (int i=1; i<=20; i++) job.add(new ExampleTask(i)).setId("task " + i);

// set the policy to execute on slaves only
ExecutionPolicy slavePolicy = new Equal("jppf.node.provisioning.slave", true);
job.getSLA().setExecutionPolicy(slavePolicy);

// submit the job and get the results
List<Task<?>> results = client.submit(job);
```

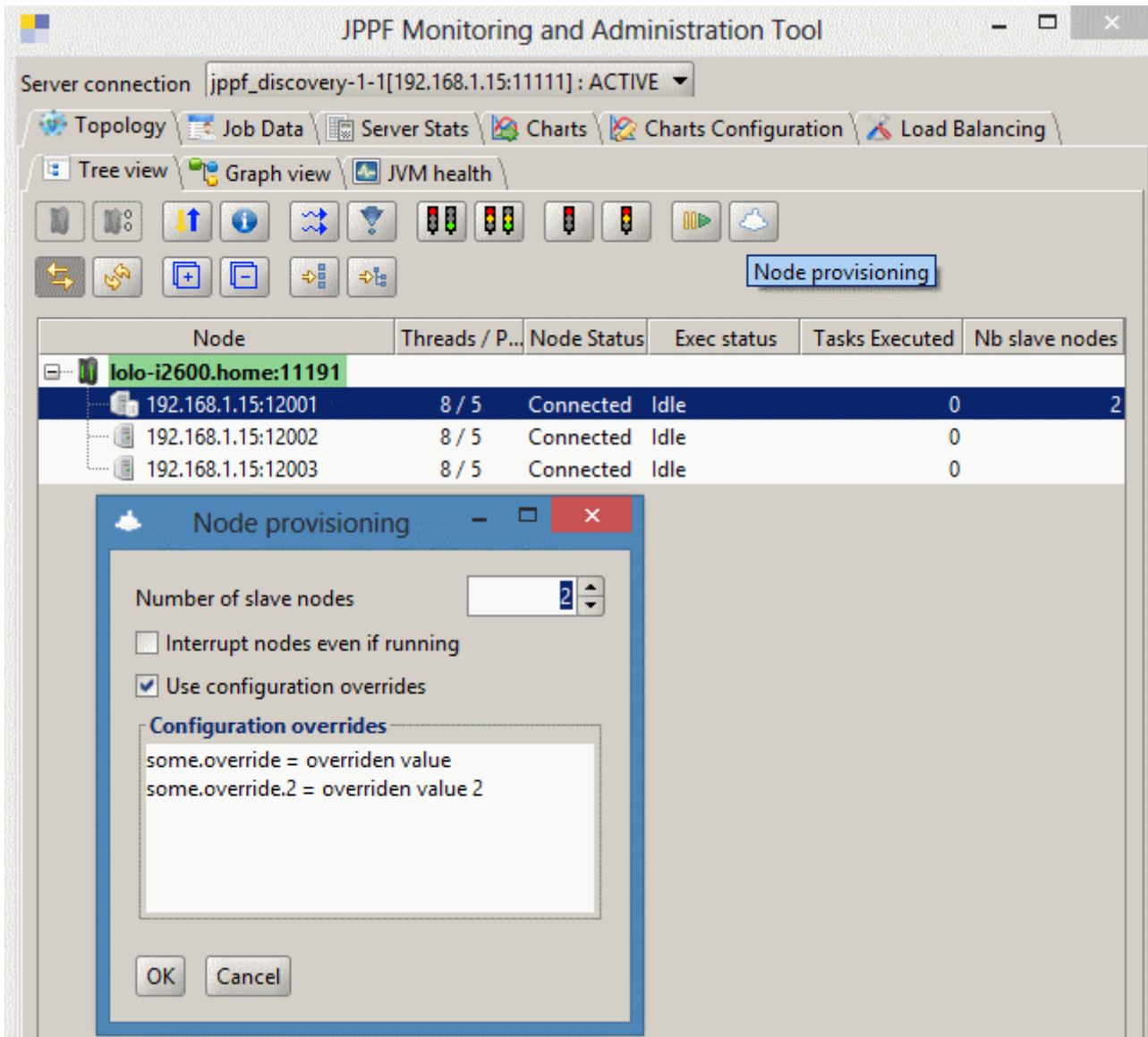
Finally, we can terminate the slave nodes with a provisioning request for 0 slaves, and get back to the initial grid topology:


```
forwarder.provisionSlaveNodes(masterSelector, 0, null);

// again, give it some time
Thread.sleep(2000L);
```




14.5.2 Provisioning with the administration console

In the JPPF administration console, the provisioning facility is available as shown here:



As you can see in this screenshot, the provisioning facility is integrated in the JPPF administration tool. To perform a provisioning operation, in the topology tree or graph view, select any number of master nodes, then click on the provisioning button  in the tool bar or in the mouse popup menu.

The topology tree view has a column indicating the number of slaves started for each master node. Non-master nodes have an empty value in this column, whereas master nodes have a value of zero or more. Furthermore, master and non-master nodes have distinctive icons:

-  for master nodes
-  for non-master nodes

After clicking on the provisioning button, a dialog will be displayed, allowing you to specify the number of slave nodes to provision, whether to use configuration overrides, and specify the overrides in free-form text. When clicking on the “OK” button, the provisioning request will be sent to all the selected master nodes.

Please note that the values entered in the provisioning dialog, including the state of the checkbox, are persisted by the administration tool, so that you will conveniently retrieve them the next time you open the dialog.

14.5.3 Configuration

14.5.3.1 Provisioning under the hood

Before a slave node is started, the master node will perform a number of operations, to ensure that the slave is properly configured and that any output it produces can be captured and retrieved. These operations include:

- 1) Creating a root directory for the slave, in which log files and output capture files will be created, along with configuration files. By default, this directory is in “`${MASTER_ROOT}/slave_nodes/node_nn`”, where the suffix *nn* is a sequence number assigned to the slave by the master node.
- 2) Copy the content of a user-specified directory, holding configuration files used by all the slaves, into the slave's “config” directory. For example, if “`slave_config`” is specified by the user, then all the files and sub-directories contained in the folder “`${MASTER_ROOT}/slave_config`” will be copied into “`${SLAVE_ROOT}/config`”. Note that the destination folder name “`config`” cannot be changed.
- 3) The JPPF configuration properties of the master node will be saved into a properties file located in “`${SLAVE_ROOT}/config/jppf-node.properties`”, after the user-specified (via the management API or the administration tool) overrides are applied, then the following overrides:

```
# mark the node as a slave
jppf.node.provisioning.slave = true
# a slave node cannot be a master
jppf.node.provisioning.master = false
# redirect the output of System.out to a file
jppf.redirect.out = system_out.log
# redirect the output of System.err to a file
jppf.redirect.err = system_err.log
```

- 4) The classpath of the slave node will be exactly the same as for the master, with the addition of the slave's root directory. This means that any jar file or class directory specified in the master's start command will also be available to the slaves.

- 5) Additional JVM options for the slave process can be specified in two ways:
 - first by overriding the “`jppf.jvm.options`” configuration property when provisioning slaves nodes
 - then, if the property “`jppf.node.provisioning.slave.jvm.options`” is defined in the master node, these options are added

For instance, setting “`jppf.node.provisioning.slave.jvm.options = -Dlog4j.configuration=config/log4j.properties`” will ensure that each slave will be able to find the Log4j configuration file in its “`config`” folder.

- 6) The master node maintains a link with each of its slaves, based on a local TCP socket connection, which serves two essential purposes:

- even when a slave is forcibly terminated, the master will know almost immediately about it and will be able to update its internal state, for instance the number of slaves
- when a master is forcibly terminated (e.g. with ‘kill -9 pid’ on Linux, or with the Task Manager on Windows), all its slaves will know about it and terminate themselves, to avoid having hanging Java processes on the host

14.5.3.2 Configuration properties

The following properties are available:

```
# Define a node as master. Defaults to true
jppf.node.provisioning.master = true
# Define a node as a slave. Defaults to false
jppf.node.provisioning.slave = false
# Specify the path prefix used for the root directory of each slave node
# defaults to "slave_nodes/node_", relative to the master root directory
jppf.node.provisioning.slave.path.prefix = slave_nodes/node_
# Specify the directory where slave-specific configuration files are located
# Defaults to the "config" folder, relative to the master root directory
jppf.node.provisioning.slave.config.path = config
# A set of space-separated JVM options always added to the slave startup command
jppf.node.provisioning.slave.jvm.options = -Dlog4j.configuration=config/log4j.properties
# Number of slaves to start at master node startup. Defaults to 0
jppf.node.provisioning.startup.slaves = 5
```

Note that “`jppf.node.provisioning.slave`” is only used by slave nodes and is always ignored by master nodes.

Additionally, each slave node has a property **"jppf.node.provisioning.master.uuid"**, whose value is the uuid of the master node that started it. This can be very useful when using a node selector or execution policy that only selects the slave nodes of one or more specific master nodes:

```
String masterUuid = ...;
JPPFJob job = new JPPFJob();
// execute this job only on slave nodes of the specified master
ExecutionPolicy policy = new Equal(
    "jppf.node.provisioning.master.uuid", true, masterUuid);
job.getSLA().setExecutionPolicy(policy);
```

14.6 Runtime dependencies

14.6.1 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
JPPF common	jppf-common.jar	Common JPPF APIs and utilities used in all components
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
JPPF JMX remote	jppf_jmxremote_optional-x.y.jar	JMX remote connector
Apache Commons IO	commons-io-2.4.jar	Apache Commons IO library

14.6.2 Node

[Common dependencies](#) +

Library name	Jar Files	Comments
JPPF node	jppf-node.jar	Node bootstrapping, distributed class loader and other node APIs

14.6.3 Driver dependencies

[Common dependencies](#) +

Library name	Jar Files	Comments
JPPF server	jppf-server.jar	Driver-specific code
JPPF node	jppf-node.jar	Node bootstrapping, distributed class loader and other node APIs

14.6.4 Client application dependencies

[Common dependencies](#) + application-specific dependencies +

Library name	Jar Files	Comments
JPPF client	jppf-client.jar	Client-specific code

14.6.5 Administration console dependencies

[Common dependencies](#) +

Library name	Jar Files	Comments
JPPF admin	jppf-admin.jar	Administration console-specific code
JPPF client	jppf-client.jar	Client-specific code
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

14.7 Web administration console

14.7.1 What is it?

The JPPF web administration console is a web-based functional equivalent to the desktop-based console, with a few significant differences:

- it has all the features of its desktop counterpart, except for the charts, which are not implemented yet
- it is multi-user and handles both authentication and authorizations
- its configuration is manageable from its own administrative user interface

The web administration console is based on the [Apache Wicket 7.x](#) web framework, which requires a servlet container supporting the [servlet 3.0 specification](#). As for the current version of JPPF, Java 7 or later is also required.

14.7.2 Authentication and authorizations

14.7.2.1 Authentication

The web administration uses form-based, container-managed authentication. This means that user credentials (user name and password) are entered via a login screen and then validated by the web-or application server, using the mechanisms exposed by the server.

14.7.2.2 User roles

The web administration console defines 3 user roles, each with its own set of associated authorizations:

- **jppf-monitor**: users in this role can only observe the state of the grid and of the jobs that run in it. They cannot stop/restart drivers or nodes, change their configuration, start slave nodes, suspend or cancel jobs, etc. Actions that are not permitted will always be disabled in the toolbars. Pages that are not allowed will not be displayed in the navigation bar on the left side.
- **jppf-manager**: users in this role have all the permissions of the "jppf-monitor" role along with the rights to change the state and behavior of the grid, including drivers, nodes and jobs.
- **jppf-admin**: users in this role have access to an administration screen not visible to other roles, which allows them to change, import and export the JPPF configuration of the web admin console and to restart the underlying JPPF client to take configuration changes into account. A user with only this role will only see the administration screen. Therefore, it is recommended to use it in combination with one of the other two roles.

14.7.2.3 User roles mapping

The task of mapping users or groups to application roles depends almost entirely on the application server in which the web console is deployed. For those servers that require a specific deployment constructor to specify roles mapping (currently Glasfish and Weblogic), such a descriptor is provided and included in the .war file. In any case, you should follow the specific server's documentation to define a user registry and/or map it to the JPPF roles.

14.7.3 Supported application servers

The JPPF web admin console has been tested on Tomcat 7.0, Jetty 9.2, Glasfish 4.1, Wildfly 8.2, Websphere Liberty Profile 16.0.0.3 and Weblogic 12.1.3. It is designed to work on any servlet container that supports the Servlet 3.0 specification.

14.7.4 Building from the source

The JPPF web administration console is distributed as a self-contained module and includes sources, required libraries, deployment descriptors, resources and build scripts. As for any JPPF module, the web console is built with [Apache Ant](#). It is recommended to have Ant 1.9.0 or later installed before building.

To build, simply unzip the JPPF-x.y.z-admin-web.zip file into a directory of your choice, then open a shell in the resulting **JPPF-x.y.z-admin-web** folder. In this shell type "ant build" or just "ant" to build the JPPFWebAdmin.war file. The file is created in the **build/** sub-folder. The application resources, especially the deployment descriptors, are found in the **webapp/** sub-folder.

14.8 Embedded driver and node

14.8.1 Embedded driver

A JPPF driver can be started programmatically using the [JPPFDriver](#) class, defined as:

```
public class JPPFDriver {
    // Initialize this JPPF driver with the specified configuration
    public JPPFDriver(TypedProperties configuration)

    // Initialize and start this driver
    public JPPFDriver start() throws Exception

    // Get this driver's unique identifier
    public String getUuid()

    // Get a server-side representation of a job from its uuid
    public JPPFDistributedJob getJob(String jobUuid)

    // Get the object which manages the job dispatch listeners
    public JobTasksListenerManager getJobTasksListenerManager()

    // Get the system information for this driver
    public JPPFSystemInformation getSystemInformation()

    // Get the object holding the statistics monitors
    public JPPFStatistics getStatistics()

    // Add a custom peer driver discovery mechanism to those already registered, if any
    public void addDriverDiscovery(PeerDriverDiscovery discovery)

    // Remove a custom peer driver discovery mechanism from those already registered
    public void removeDriverDiscovery(PeerDriverDiscovery discovery)

    // Determine whether this server is shutting down,
    // in which case it does not accept connections anymore
    public boolean isShuttingDown()

    // Get this driver's configuration
    public TypedProperties getConfiguration()

    // Shutdown this driver and all its components
    public void shutdown()
}
```

The driver's life cycle is as follows:

- 1) create it using the constructor `JPPFDriver(TypedProperties)` which takes a [configuration](#) object
- 2) start the driver using the `start()` method
- 3) when it is no longer needed, shut it down with the `shutdown()` method

Here is an example usage:

```
// create the driver configuration
TypedProperties driverConfig = new TypedProperties()
    .set(JPPFProperties.SERVER_PORT, 11111)
    // disable SSL
    .set(JPPFProperties.SSL_SERVER_PORT, -1);

// start the JPPF driver
final JPPFDriver driver = new JPPFDriver(driverConfig).start();

// ... use the driver ...

// now shut it down
driver.shutdown();
```

14.8.2 Embedded node

A JPPF node can be created and started using the [NodeRunner](#) class, which will create a [Node](#) object. [NodeRunner](#) exposes the following API:


```

public class NodeRunner {
    // Initialize this node runner with the specified configuration
    public NodeRunner(TypedProperties configuration)

    // Run a node embedded in the current JVM
    public void start(String...args)

    // Shutdown the node
    public void shutdown()

    // Get the node's universal unique identifier
    public String getUuid()

    // Determine whether the node is offline
    public boolean isOffline()

    // Get the actual node started by this node runner, if any
    public Node getNode()
}

```

From this API, the node life cycle is straightforward:

- 1) create a NodeRunner using the constructor NodeRunner(TypedProperties) which take a [configuration](#) object
- 2) start the node with the start() method

Important note: the start() method is blocking, therefore it should be called from a separate thread to avoid blocking the current thread, as in this example:

```

final NodeRunner runner = ...;
new Thread(() -> runner.start()).start();

```

- 3) when the node is no longer needed, shut it down by calling shutdown()

Note: the [Node](#) instance returned by getNode() may vary over time: whenever the node is disconnected from the driver, the NodeRunner will create a new Node instance and attempt to connect it back.

Example usage:

```

// create the node configuration
TypedProperties nodeConfig = new TypedProperties()
    .set(JPPFProperties.SERVER_HOST, "localhost")
    .set(JPPFProperties.SERVER_PORT, 11111)
    .set(JPPFProperties.MANAGEMENT_PORT, 11111)
    .set(JPPFProperties.SSL_ENABLED, false);

final NodeRunner nodeRunner = new NodeRunner(nodeConfig);
// start the node in a separate thread
new Thread(() -> nodeRunner.start()).start();

// ... use the node ...

// shut down the node
nodeRunner.shutdown();

```

14.8.3 Related sample

Please refer to the [Embedded Grid](#) demo.

14.9 JPPF in Docker containers

14.9.1 Docker images and registry

The latest JPPF docker images are available on [Docker Hub](#) in the [JPPF docker registry](#)

We currently have images and corresponding [Dockerfiles](#) for:

- JPPF drivers
- JPPF nodes
- the JPPF web administration console

14.9.2 Deployment on Kubernetes

JPPF docker images can be deployed in a Kubernetes cluster using the [JPPF helm chart](#).

JPPF hosts its own Helm charts repository at <https://www.jppf.org/helm-charts>. To add this repository under the name "jppf-repo":

```
$ helm repo add jppf-repo https://www.jppf.org/helm-charts
```

To list all the charts in the JPPF repository:

```
$ helm search jppf-repo/
```

To install from the repository:

```
$ helm install --name jppf --namespace jppf jppf-repo/jppf
```

To uninstall the JPPF cluster:

```
$ helm delete jppf
```

For more details on configuration and usage, the [full helm chart documentation](#) is available in the source code repository.

14.9.3 Deployment on Docker swarm

JPPF can also be deployed in a Docker swarm cluster with Docker stack. The deployment relies on a [docker-compose.yml](#) file which defines the JPPF services to deploy, and a [.env](#) file which defines the environment variables the services depend on.

To deploy JPPF in a swarm cluster with a stack name "jppf":

```
$ docker stack deploy -c ./docker-compose.yml jppf
```

To shutdown the JPPF service stack:

```
$ docker stack rm jppf
```

15 Changes from previous versions

15.1 Changes in JPPF 6.0

15.1.1 Package org.jppf.client

In the class `AbstractJPPFClient`, the following deprecated methods were removed:

- `addClientListener(ClientListener)`
- `removeClientListener(ClientListener)`
- `getAllConnectionNames()`
- `getAllConnections()`
- `getClientConnection()`
- `getClientConnection(JPPFClientConnectionStatus...)`
- `getClientConnection(String)`

In the class `JPPFClient`, the following deprecated constructors were removed:

- `JPPFClient(ClientListener...)`
- `JPPFClient(String, ClientListener...)`
- `JPPFClient(String, TypedProperties, ClientListener...)`

In the enum `Operator`, the following deprecated enum constants were removed:

- `GREATER`
- `LESS`

In the interface `ConnectionPool`, the following deprecated methods were removed:

- `getMaxSize()`
- `setMaxSize(int)`

In the class `AbstractConnectionPool`, the following deprecated methods were removed:

- `getMaxSize()`
- `setMaxSize(int)`

15.1.2 Package org.jppf.client.event

The following deprecated classes were removed:

- `ClientListener`
- `ClientEvent`

15.1.3 Package org.jppf.management

The following deprecated classes were removed:

- `NodeSelector.AllNodesSelector`
- `NodeSelector.ExecutionPolicySelector`
- `NodeSelector.UuidSelector`

15.1.4 Package org.jppf.job

The following deprecated classes were removed:

- `TaskReturnEvent`
- `TaskReturnListener`
- `TaskReturnManager`

15.1.5 Package org.jppf.load.balancer

In interface LoadBalancingProfile, the deprecated `copy()` method was removed

In interface Bundler, the deprecated `copy()` method was removed

In class AbstractLoadBalancingProfile, the deprecated `copy()` method was removed

In class AbstractBundler, the deprecated `copy()` method was removed

The interface JobAwarenessEx was renamed to JobAwareness and JobAwarenessEx was removed

The deprecated NodeAwareness interface was removed

15.1.6 Package org.jppf.load.balancer.impl

The following deprecated classes were removed:

- RLBundler
- RLProfile

15.1.7 Package org.jppf.load.balancer.spi

The deprecated class RLBundlerProvider was removed.

15.1.8 Package org.jppf.node.screensaver

In class NodeIntegrationAdapter the deprecated method `setScreenSaver(JPPFScreenSaver)` was removed

In interface NodeIntegration the deprecated method `setScreenSaver(JPPFScreenSaver)` was removed

15.1.9 Package org.jppf.server.job.management

In class DriverJobManagementMBean the deprecated method `getAllJobIds()` was removed

15.1.10 Package org.jppf.node.protocol

In interface Task and class AbstractTask:

- the deprecated method `getException()` was removed
- the method `Node getNode()` was added

15.1.11 Package org.jppf.node

In class Node, the method `resetTaskClassLoader(Object... params)` now returns a ClassLoader object instead of an AbstractJPPFClassLoader

15.1.12 Package org.jppf.utils

The following classes were moved to the new package org.jppf.utils.concurrent

- ConcurrentUtils
- JPPFThreadFactory
- MutableReference
- ThreadSynchronization

15.2 Changes in JPPF 5.0

15.2.1 New packaging

Version 5.0 introduces a more consistent packaging and distribution of the JPPF libraries. The following table compares the JPPF libraries present in each component distribution between versions 5.0 and 4.0:

Distribution	Jars in 5.0	Jars in 4.0
Node	jppf-common.jar jppf-node.jar	jppf-common-node.jar
Server	jppf-common.jar jppf-node.jar jppf-server.jar	jppf-common-node.jar jppf-common.jar jppf-server.jar
Client	jppf-common.jar jppf-client.jar	jppf-common-node.jar jppf-common.jar jppf-client.jar
Administration console	jppf-common.jar jppf-client.jar jppf-admin.jar	jppf-common-node.jar jppf-common.jar jppf-client.jar jppf-admin.jar

15.2.2 Location API classes:

The following classes were moved from the package `org.jppf.node.protocol` to the package `org.jppf.location`:

- `AbstractLocation`
- `FileLocation`
- `Location`
- `LocationEvent`
- `LocationEventListener`
- `MemoryLocation`
- `URLLocation`

15.2.3 Load-balancer classes

The classes which were in the package `org.jppf.server.schedule.bundle` and its subpackages were moved to the `org.jppf.load.balancer` package and subpackages:

v4.0 packages	Moved to in v5.0
<code>org.jppf.server.scheduler.bundle</code>	<code>org.jppf.load.balancer</code>
<code>org.jppf.server.scheduler.bundle.autotuned</code> <code>org.jppf.server.scheduler.bundle.fixedsize</code> <code>org.jppf.server.scheduler.bundle.impl</code> <code>org.jppf.server.scheduler.bundle.nodethreads</code> <code>org.jppf.server.scheduler.bundle.proportional</code> <code>org.jppf.server.scheduler.bundle.rl</code>	<code>org.jppf.load.balancer.impl</code>
<code>org.jppf.server.scheduler.bundle.providers</code> <code>org.jppf.server.scheduler.bundle.spi</code>	<code>org.jppf.load.balancer.spi</code>

Accordingly, the service definition file `META-INF/services/org.jppf.server.schedule.bundle.spi.JPPFBundlerProvider` was moved to `META-INF/services/org.jppf.load.balancer.spi.JPPFBundlerProvider`

15.2.4 Package org.jppf.server.protocol

The following classes were moved to the package org.jppf.node.protocol:

- JPPFTask
- JPPFRunnable
- CommandLineTask
- ClassPathElementImpl

15.2.5 Package org.jppf.task.storage

The following classes were moved to the package org.jppf.node.protocol:

- DataProvider
- MemoryMapDataProvider

15.2.6 Package org.jppf.client

class JPPFJob: the method getStatus() now returns an org.jppf.client.JobStatus enum value

15.2.7 Package org.jppf.client.submission

This package has been removed.

The class SubmissionStatus was moved and renamed as org.jppf.client.JobStatus

15.2.8 Package org.jppf.client.event

Class SubmissionStatusEvent:

- the class was renamed to JobStatusEvent
- the method getStatus() now returns an org.jppf.client.JobStatus enum value
- the method getSubmissionId() was renamed to getJobUuid()

Class SubmissionStatusListener:

- the class was renamed to JobStatusListener
- the method submissionStatusChanged(SubmissionStatusEvent) was refactored into jobStatusChanged(JobStatusEvent)

15.2.9 Package org.jppf.jca.cci

Interface JPPFConnection:

- submit(JPPFJob job, SubmissionStatusListener) was refactored into submit(JPPFJob, JobStatusListener)
- addSubmissionStatusListener(String, SubmissionStatusListener) was refactored into addJobStatusListener(String, JobStatusListener)
- removeSubmissionStatusListener(String, SubmissionStatusListener) was refactored into removeJobStatusListener(String, JobStatusListener)
- SubmissionStatus getSubmissionStatus(String) was refactored into JobStatus getJobStatus(String)
- getAllSubmissionIds() was renamed to getAllJobIds()

15.3 API changes in JPPF 4.0

15.3.1 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.

15.3.2 The new Task API

15.3.2.1 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 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 officially [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.

15.3.2.2 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 specific exception type.

A new method `fireNotification(Object userObject, boolean sendViaJmx)` was added, which allows the tasks to send notifications to local listeners or remote listeners via JMX.

15.3.3 Package `org.jppf.client`

15.3.3.1 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.

15.3.3.2 Class `AbstractJPPFClient`

As for `JPPFClient`, the abstract definition of `submit(JPPFJob)` is deprecated and replaced with the method `submitJob(JPPFJob)`,

15.3.3.3 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<?>`.

15.3.3.4 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<?>>`.

15.3.3.5 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<?>>)`

15.3.4 Package `org.jppf.client.event`

15.3.4.1 Class *JobEvent*

`getTasks()` : `List<JPPFTask>` is deprecated and replaced with `getJobTasks()` : `List<Task<?>>`.

15.3.4.2 Class *TaskResultEvent*

`getTaskList()` : `List<JPPFTask>` is deprecated and replaced with `getTasks()` : `List<Task<?>>`.

15.3.5 Package `org.jppf.client.persistence`

15.3.5.1 Class *JobPersistence*

`storeJob(K, JPPFJob, List<JPPFTask>)` was changed into `storeJob(K, JPPFJob, List<Task<?>>)`.

15.3.5.2 Class *DefaultFilePersistenceManager*

`storeJob(K, JPPFJob, List<JPPFTask>)` was changed into `storeJob(K, JPPFJob, List<Task<?>>)`.

15.3.6 Package `org.jppf.client.taskwrapper`

The class `JPPFTaskCallback` has been genericized into `JPPFTaskCallback<T>`.
`getTask()` : `JPPFTask` was changed into `getTask()` : `Task<T>`.

15.3.7 Package `org.jppf.client.utils`

15.3.7.1 Class *ClientWithFailover*

`submit(JPPFJob)` : `List<JPPFTask>` is deprecated and replaced with `submitJob(JPPFJob)` : `List<Task<?>>`.

15.3.8 Package `org.jppf.server.protocol`

15.3.8.1 Class *CommandLineTask* genericized

`CommandLineTask` is now defined as `CommandLineTask<T>` extends `AbstractTask<T>`.

15.3.9 Package `org.jppf.node.event`

The classes `TaskExecutionListener` and `TaskExecutionEvent` moved from package `org.jppf.server.node`.

15.3.9.1 Class *TaskExecutionListener*

The method `taskNotification(TaskExecutionEvent)` has been added.

15.3.9.2 Class *TaskExecutionEvent*

The method `getTask()` : `JPPFTask` was changed into `getTask()` : `Task<?>`.

15.3.10 Package `org.jppf.jca.cci` (J2E connector)

in 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.