



ProActive
Programming, Composing, Deploying on the Grid



ProActive Scheduler

OASIS Research Team and ActiveEon Team



Version 2008-07-16 10:47:41 2008-07-16 10:47:41

Copyright © 1997-2008 INRIA

ProActive Scheduler v2008-07-16 10:47:41 Documentation

ProActive Scheduler

OASIS Research Team and ActiveEon Team

Legal Notice

The ProActive Scheduler is being distributed under the GPL2 license.

Copyright INRIA 1997-2008.

Contributors and Contact Information

Team Leader:

Denis Caromel
INRIA 2004, Route des Lucioles
BP 93
06902
Sophia Antipolis Cedex
France
phone: +33 492 387 631
fax: +33 492 387 971
Denis.Caromel@inria.fr

ActiveEon Team

Christian Delbé
Arnaud Contes
Vladimir Bodnartchouk
Emil Salageanu

OASIS Team

Guillaume Laurent	Baptiste De Stefano
Robert Lovas	Nicolas Dodelin
Jonathan Martin	Yu Feng
Elton Mathias	Imen Filiali
Maxime Menant	Johann Fradj
Guilherme Perretti Pezzi	Abhijeet Gaikwad
Franca Perrina	Regis Gascon
Kamran Qadir	Jean-Michael Guillamume
Bastien Sauvan	Abhishek-Rajeev Gupta
Germain Sigety	Elaine Isnard
Etienne Vallette-De-Osia	Vasile Jureschi
Laurent Vanni	Francoise Baude
Yulai Yuan	Antonio Cansado
Sylvain Cussat-Blanc	Marcela Rivera
Boutheina Bennour	Ludovic Henrio
Vincent Cave	Fabrice Huet
Guillaume Chazarain	Virginie Contes
Clement Mathieu	Mario Leyton
Eric Madelaine	Paul Naoumenko
Brian Amedro	Viet Dong Doan
Florin Bratu	Fabien Viale
Tomasz Dobek	Cédric Dalmaso
Khan Muhammad	Jean-Luc Scheefer
Julian Krzeminski	
Zhihui Dai	

Past And External Important Contributors

Lionel Mestre	Laurent Baduel
Matthieu Morel	Alexandre di Costanzo
Guillaume Chazarain	Romain Quilici
	Nadia Ranaldo
	Julien Vayssiere

Public questions, comments, or discussions can be posted on the ProActive public mailing list

proactive@ow2.org

The mailing list archive is placed at

<http://www.objectweb.org/www/arc/proactive>

Bugs can be posted on the ProActive Jira bug-tracking system

<https://galpage-exp.inria.fr:8181/jira>

Table of Contents

List of figures	iv
List of examples	v

Part I. ProActive Scheduler

Chapter 1. ProActive Scheduler	2
1.1. IMPORTANT NOTE	2
1.2. Overview	2
1.3. Scheduler Concept	2
1.3.1. What is a Job ?	2
1.3.2. What is a Task ?	2
1.3.3. Dependencies between Tasks	3
1.3.4. Scheduling Policy	4
1.4. User Manual	4
1.4.1. Create a job	4
1.4.2. Create a TaskFlow job	5
1.4.3. Create a ProActive job	6
1.4.4. Create and Add a task to a job	7
1.4.5. Submit a job to the ProActive Scheduler	22
1.4.6. Get a Job result	23
1.4.7. Register to ProActive Scheduler events	23
1.5. Administrator Manual	24
Chapter 2. ProActive Scheduler Eclipse Plugin	25
2.1. The Scheduler perspective	25
2.2. Views composing the perspective	27
2.3. Connect to the started ProActive Scheduler	31
2.4. The Scheduler perspective buttons	32
2.4.1. The Jobs view buttons in User Mode	32
2.4.2. The Jobs view buttons in Administrator Mode	33

Part II. ProActive Resource Manager

Chapter 3. ProActive Resource Manager	36
3.1. IMPORTANT NOTE	36
3.2. Role	36
3.3. Resource Manager architecture	36
3.4. Static Node Source and Dynamic Node Source	37
3.5. Nodes states	37
3.6. Starting the Resource Manager	38
Chapter 4. Resource Manager's Eclipse Plugin	39

Part III. ProActive Scheduler's Matlab extension

Chapter 5. ProActive Scheduler's Matlab Extension	41
5.1. Presentation	41

5.2. Quick Start with the Matlab Extension	41
5.2.1. Installation	41
5.2.2. Writing a simple example : the Matlab Script	41
5.2.3. Writing a simple example : the Scheduler job descriptor	41
5.3. A More Complex Example : a Matlab task flow	45
5.3.1. Descriptor variables	47
5.3.2. New Tasks : MatlabSplitter and MatlabCollector	47
5.3.3. Task dependencies	47
5.3.4. New parameter in SimpleMatlab tasks: index	47
5.3.5. Matlab Scripts for this example	48

Part IV. ProActive Scheduler's Scilab extension

Chapter 6. ProActive Scheduler's Scilab Extension	50
6.1. Presentation	50
6.2. Quick Start with the Scilab Extension	50
6.2.1. Installation	50
6.2.2. The Scilab Job descriptor	50

List of Figures

1.1. Task flow job example	3
1.2. CancelOnError and RestartOnError behavior	21
2.1. The Scheduler Perspective	26
2.2. The Jobs view	27
2.3. The Console view	28
2.4. The Tasks view	29
2.5. The Job Info view	30
2.6. The Result Preview view	31
2.7. Connect to scheduler	31
2.8. Connect to scheduler	32
2.9. Disconnect from the scheduler	32
2.10. Change view from Vertical to Horizontal mode	32
2.11. Change view from Horizontal to Vertical mode	32
2.12. Submit a job	32
2.13. Pause/Resume a job	32
2.14. Change job priority	32
2.15. Display job output	32
2.16. Kill Job	33
2.17. Start the scheduler	33
2.18. Stop the scheduler	33
2.19. Freeze the scheduler	33
2.20. Pause the scheduler	33
2.21. Resume the scheduler	33
2.22. Shutdown the scheduler	33
2.23. Kill scheduler	33
3.1. resource Manager architecture	37

List of Examples

5.1. Simple Matlab Job descriptor Example	44
5.2. Complex Matlab Job descriptor Example	46
6.1. Scilab Job descriptor Example	51
6.2. Integral script	52
6.3. Merging script	52

Part I. ProActive Scheduler

Table of Contents

Chapter 1. ProActive Scheduler	2
1.1. IMPORTANT NOTE	2
1.2. Overview	2
1.3. Scheduler Concept	2
1.3.1. What is a Job ?	2
1.3.2. What is a Task ?	2
1.3.3. Dependencies between Tasks	3
1.3.4. Scheduling Policy	4
1.4. User Manual	4
1.4.1. Create a job	4
1.4.2. Create a TaskFlow job	5
1.4.3. Create a ProActive job	6
1.4.4. Create and Add a task to a job	7
1.4.5. Submit a job to the ProActive Scheduler	22
1.4.6. Get a Job result	23
1.4.7. Register to ProActive Scheduler events	23
1.5. Administrator Manual	24
Chapter 2. ProActive Scheduler Eclipse Plugin	25
2.1. The Scheduler perspective	25
2.2. Views composing the perspective	27
2.3. Connect to the started ProActive Scheduler	31
2.4. The Scheduler perspective buttons	32
2.4.1. The Jobs view buttons in User Mode	32
2.4.2. The Jobs view buttons in Administrator Mode	33

Chapter 1. ProActive Scheduler

1.1. IMPORTANT NOTE

- Some parts of the ProActive Scheduler and ProActive Resource Manager rely on Java Scripting capabilities (JSR 223 [<http://jcp.org/en/jsr/detail?id=223>]). As a consequence, it requires either:

- a 1.6 or greater Java Runtime Environment, without any modifications,
- or, with a 1.5 JRE, the JSR 223 jar files [<http://jcp.org/aboutJava/communityprocess/final/jsr223/index.html>] :
 - First, the `script-api.jar`, `script-js.jar` and `js.jar` files must be added in the `/ProActive/dist/lib/` directory if you are using the bin release or ProActive, or in the `/ProActive/lib/` directory if you build ProActive from the source release.
 - Then the `java5_jsr223_patch.jar` patch (released with the Scheduler RCP Client) should be executed in the Scheduler RCP Client directory : unzip the `java5_jsr223_patch.zip` file and execute `java -jar java5_jsr223_patch.jar` .

1.2. Overview

The execution of parallel tasks on a pool of distributed resources (what we call 'nodes'), such as network of desktops or clusters, requires a main system for managing resources and handling task execution: a **batch scheduler** . A batch scheduler provides an abstraction of resources to users. Users submit jobs containing tasks to the **scheduler** , who is in charge of executing these tasks on the resources. A **scheduler** allows several users to share a same pool of resources and also to manage all issues related to distributed environment, such as faulted resources. The ProActive Scheduler is connected to a resource manager that will do the resource abstraction.(see Chapter 3, *ProActive Resource Manager*)

In this chapter we present a ProActive based Scheduler accessible either from a **Java programming API** , a **command-line based job submitter**. It is also recommended to use **the graphical user or administration interface** (Eclipse RCP Plugin, see Chapter 2, *ProActive Scheduler Eclipse Plugin*) which can be plugged on the scheduler core application.

In the rest of this chapter, we will expose how the scheduler works, what policies govern the job management, how to create a job and how to get the jobs and the nodes state using either the shell communicator or the GUI.

NOTE - Additionally, you can find here [<http://proactive.inria.fr/userfiles/file/tutorials/ProActiveSchedulerTutorial.pdf>] a fully documented example of the Scheduler and Resource Manager usage. This tutorial does not require Java nor ProActive knowledge since it's only based on graphical interface and command line actions.

1.3. Scheduler Concept

1.3.1. What is a Job ?

A **Job** is the entity to be submitted to the scheduler. It is composed of one or more **Tasks** . A Job can have one of the following types :

- **TASKSFLOW** , represents a Job that contains a bag of Tasks, which can be executed in parallel or according to a dependency tree. The Tasks inside this Job type can be either Java (A task written in Java extending a given interface) or Native (Any native process).
- **PROACTIVE** , represents a Job that contains a ProActive application (embedded in **only one ProActive Task**). Its execution starts with a given predefined number of resources on which the user can start the ProActive application. This kind of Job requires the usage of the ProActive API, in order to be able to build ProActive application.

A finished Job contains a result that is provided by the scheduler once the job terminated, which in term contains all of its tasks' results. However, it is possible to mark some task as **precious** in order to retrieve their result easily in the job result. In the event of a failure, the finished Job contains the causes of the exception. Further details on how to create a Job and the different options can be found in: Section 1.4.1, "Create a job" .

1.3.2. What is a Task ?

The **Task** is the smallest schedulable entity. It is included in a **Job** (see Section 1.3.1, "What is a Job ?") and will be executed in accordance with the scheduling policy (see Section 1.3.4, "Scheduling Policy") on the available resources.

There are three types of Tasks :

- **JAVA** ; its execution is defined by a Java class extending the `org.ow2.proactive.scheduler.common.task.executable.JavaExecutable` class.
- **NATIVE** ; its execution can be any user program specified by a simple command line, or by a 'generation script', that can dynamically generates the command line to be executed.
- **PROACTIVE** ; its execution is defined by a Java class extending the `org.ow2.proactive.scheduler.common.task.executable.ProActiveExecutable` class, which defines a ProActive application. Coding this last one requires a knowledge base on the use of ProActive. Needed resources are provided, it is no need to learn about the deployment.

During its execution, a Task can crash due to host or code failure. It's good to know that a Task can be re-started a parameterizable number of time (see re-runnable in section Section 1.4.4, "Create and Add a task to a job").

A Task may optionally be accompanied by 3 kinds of scripts (pre-script, post-script and selection-script), that allow to select the suitable resource for a given task and possibly configure it before and after task execution (see Section 1.4.4, "Create and Add a task to a job").

Dependencies between Tasks can also be defined; this aspect is detailed in next section.

1.3.3. Dependencies between Tasks

Dependencies can be set between Tasks in a TaskFlow Job. It provides a way to execute your tasks in a specified order, but also to forward the results of an ancestor task to its children as parameter. Dependency between task is then both a temporal dependency and a data dependency.

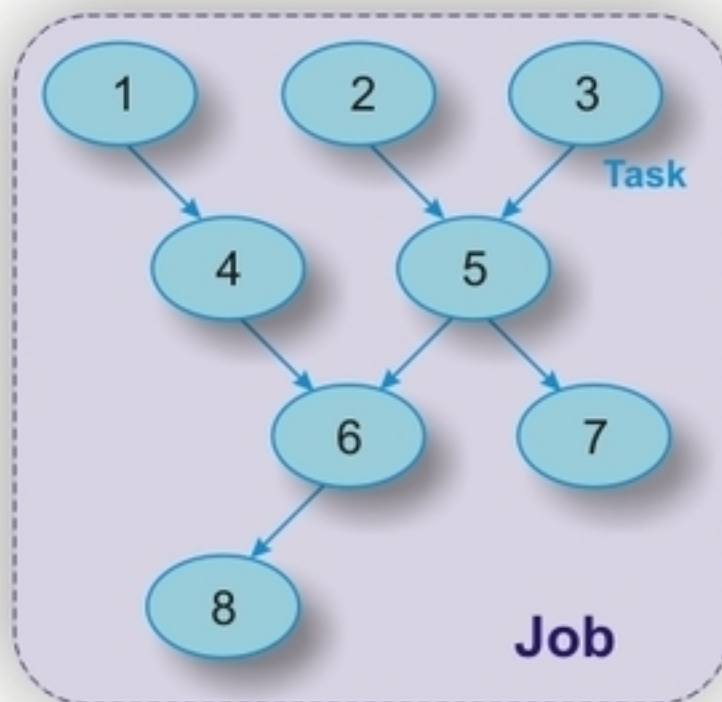


Figure 1.1. Task flow job example

In this example we made an 8 Tasks Job (where the Job's type is TaskFlow). As you can see, Task 4 depends on Task 1, Task 5 depends on Tasks 2 and 3, etc... In other words, Task 4 will wait for Task 1 to finish before starting, Task 5 will wait for Task 2

AND 3, etc... In addition, the order in which you specify that Task 5 depends of Task 2 and 3 is very important. Indeed, if you set the list of dependencies for Task 5 as : 2 then 3, the result of these two task will be given to Task 5 in this order.

1.3.4. Scheduling Policy

By default , the scheduler will schedule tasks according to the default **FIFO (First In First Out) with job priority** policy. So, if you want a job to be scheduled quickly, increase its priority, or ask your administrator for an other policy.

1.4. User Manual

1.4.1. Create a job

A job is the entity that will be submitted to the ProActive Scheduler. As it has been explained in the Section 1.3.1, “What is a Job ?”, it's possible to create more than one type of job. A job can also be created using an XML descriptor or the provided ProActive Scheduler Java API.

1.4.1.1. Create a job using XML descriptor

Just follow the example below in order to create your Job with XML description :

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:0.91"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:proactive:jobdescriptor:0.91
http://proactive.inria.fr/schemas/jobdescriptor/0.91/schedulerjob.xsd"
id="job_name" priority="normal" projectName="project_name" cancelOnError="true" logFile=
"path/to/a/log/file.log">
  <description>Job description</description>
  <variables>
    <variable name="val1" value="toto"/>
  </variables>
  <genericInformation>
    <info name="var1" value="{val1}"/>
    <info name="var2" value="val2"/>
  </genericInformation>

  <!-- Job will be completed here later --->
</job>
```

As shown, several features can be set on this job :

- **id** is a way to identify your job or just simply name it. If this value is left to the empty string, the Scheduler will set it by a default one.
- **projectName** (optional) can be define in your job. This information also goes to the policy in order to group different job by project name for example.
- **priority** (optional) is the scheduling priority level for your job. A user can only set its job priority to 'lowest', 'low', or 'normal'.
- **CancelOnError** (optional) is a way to define if your job will continue if a user exception or error occurs during the whole job process. It means that if the value of this property is true, the job will stop immediately every running task if one error occurs in one of the task of this job. It will have the consequence to failed the job, but free resources for other jobs. It is useful when it is no need to go further after a task failure.
- **logFile** (optional) is the path of an optional log file. Set it if you want to save the job generated log in a file.
- **description** (optional) is a human readable description of the job, for human use only. This field is optional but it's better to set it.
- **variables** (optional) is a way to define variables which can be reused throughout this descriptor. Inside this tag, each variable can be reused (even in another following variable definition) by using the syntax `${name_of_variable}`.

- **genericInformations** (optional) is a way to define some informations inside your job. These informations could be read by the policy of the Scheduler. It can be useful to modify the scheduling behavior. Contact your administrator if you want an information to be interpreted by the policy.

To specialize your job to a taskFlow job, go to Section 1.4.2.1, “Create a TaskFlow job using XML descriptor”.

To specialize your job to a ProActive job, go to Section 1.4.3.1, “Create a ProActive job using XML descriptor”.

1.4.1.2. Create a job using Java API

To make a new instance of a TaskFlow job, go to Section 1.4.2.2, “Create a TaskFlow job using Java API”.

To make a new instance of a ProActive job, go to Section 1.4.3.2, “Create a ProActive job using Java API”.

Then, just follow the example below in order to create your Job using the Java Scheduler API :

```
//job has already been created under the -'job' variable
job.setName("job_name");
job.setProjectName("project_name");
job.setPriority(JobPriority.NORMAL);
job.setCancelOnError(true);
job.setLogFile("path/to/a/log/file.log");
job.setDescription("Job description");
job.addGenericInformation("var1", "val1");
job.addGenericInformation("var2", "val2");
```

As shown, several features can be set on this job :

- **name** is a way to identify your job or just simply name it. If this value is left to the empty string, the Scheduler will set it by a default one.
- **projectName** (optional) can be define in your job. This information also goes to the policy in order to group different job by project name for example.
- **priority** (optional) is the scheduling priority level for your job. A user can only set its job priority to 'lowest', 'low', or 'normal'.
- **CancelOnError** (optional) is a way to define if your job will continue if a user exception or error occurs during the whole job process. It means that if the value of this property is true, the job will stop immediately every running task if one error occurs in one of the task of this job. It will have the consequence to failed the job, but free resources for other jobs. It is useful when it is no need to go further after a task failure.
- **logFile** (optional) is the path of an optional log file. Set it if you want to save the job generated log in a file.
- **description** (optional) is a human readable description of the job, for human use only. This field is optional but it's better to set it.
- **genericInformation** (optional) is a way to define some informations inside your job. These informations could be read by the policy of the Scheduler. It can be useful to modify the scheduling behavior. Contact your administrator if you want an information to be interpreted by the policy.

To create and add tasks to your Job, just go to Section 1.4.4, “Create and Add a task to a job”.

1.4.2. Create a TaskFlow job

The TaskFlowJob or data flow job is a job that can contain one or more task(s) with the dependencies you want.

To start with the job creation, please first read Section 1.4.1, “Create a job”.

1.4.2.1. Create a TaskFlow job using XML descriptor

To specify that the job is a TaskFlow Job, just add the 'taskFlow' tag. Here's an example of how to go on to a TaskFlow Job using the previous job descriptor :

```

<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:0.91"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:proactive:jobdescriptor:0.91
http://proactive.inria.fr/schemas/jobdescriptor/0.91/schedulerjob.xsd"
id="job_name" priority="normal" projectName="project_name" cancelOnError="true" logFile=
"path/to/a/log/file.log">
  <description>Job description</description>
  <variables>
    <variable name="val1" value="toto"/>
  </variables>
  <genericInformation>
    <info name="var1" value="{val1}"/>
    <info name="var2" value="val2"/>
  </genericInformation>

  <taskFlow>

    <!-- Job will be completed here later --->

  </taskFlow>
</job>

```

To create and add tasks to your Job, just go to Section 1.4.4, “Create and Add a task to a job”.

1.4.2.2. Create a TaskFlow job using Java API

To make a new instance of a TaskFlow job, just create it as shown below :

```
TaskFlowJob job = new TaskFlowJob();
```

To parameterize your TaskFlow Job, just go to Section 1.4.1.2, “Create a job using Java API”.

1.4.3. Create a ProActive job

To create a non-specialized job, please first read Section 1.4.1, “Create a job”.

1.4.3.1. Create a ProActive job using XML descriptor

To specify that the job is a ProActive Job, just add the 'proActive' tag. Here's an example of how to go on to a ProActive Job using the previous job descriptor :

```

<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:0.91"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:proactive:jobdescriptor:0.91
http://proactive.inria.fr/schemas/jobdescriptor/0.91/schedulerjob.xsd"
id="job_name" priority="normal" projectName="project_name" cancelOnError="true" logFile=
"path/to/a/log/file.log">
  <description>Job description</description>
  <variables>
    <variable name="val1" value="toto"/>
  </variables>
  <genericInformation>

```

```

<info name="var1" value="{val1}"/>
<info name="var2" value="val2"/>
</genericInformation>

<proActive neededNodes="10">

  <!-- Job will be completed here later --->

</proActive>
</job>

```

To create and add tasks to your Job, just go to Section 1.4.4, “Create and Add a task to a job”.

1.4.3.2. Create a ProActive job using Java API

To make a new instance of a ProActive job, just create it as shown below :

```
ProActiveJob job = new ProActiveJob();
```

To parameterize your ProActive Job, just go to Section 1.4.1.2, “Create a job using Java API”.

1.4.4. Create and Add a task to a job

As it has been said, it is possible to create 3 types of tasks. Native and Java tasks can be add to TaskFlow Job, and one ProActive Task to one ProActive Job.

1.4.4.1. Create and Add a Java task

Note : It is only possible to add a Java task in a TaskFlow Job.

To learn how to create a TaskFlow Job, just go to Section 1.4.2, “Create a TaskFlow job”. Once your TaskFlow Job created, you can add as many Java tasks as needed to perform an application.

1.4.4.1.1. Define your own Java executable

First of all, you must know that you can create your own java executable by implementing scheduler executable interfaces. What is called 'executable' is in fact, the executed process (that is a Java class in this case). Here's an example to create your own Java executable :

```

public class WaitAndPrint extends JavaExecutable {

    @Override
    public Object execute(TaskResult... results) throws Throwable {
        String message;

        try {
            System.err.println("Démarrage de la tache WaitAndPrint");
            System.out.println("Parameters are -: -");

            for (TaskResult tRes -: results) {
                if (tRes.hasException()) {
                    System.out.println("\t -" + tRes.getTaskId() + " -: -" + tRes.getException().getMessage());
                } else {
                    System.out.println("\t -" + tRes.getTaskId() + " -: -" + tRes.value());
                }
            }
        }
    }
}

```

```

    }
}

message = URIBuilder.getLocalAddress().toString();
Thread.sleep(10000);

} catch (Exception e) {
    message = "crashed";
    e.printStackTrace();
}

System.out.println("Terminaison de la tache");

return (message + "\t slept for 10 sec");
}
}

```

This executable will print an initial message, then check if there are results from previous tasks and if so, print the value of these "parameters". It will then return a message containing what the task did. The return value will be store in the job result.

It is also possible to get a list of arguments that you can give to the executable at its start by overriding the `init` method on a Java executable. How to give arguments to the task will be explain further. We get back the `foo`, `bar` and `test` arguments to illustrate the task creation example below :

```

private boolean foo;
private int bar;
private String test;

@Override
public void init(Map<String, Object> args) {
    foo = (Boolean)args.get("foo");
    bar = (Integer)args.get("bar");
    test = args.get("test");
}

```

To sum up, create an executable is just extend the `JavaExecutable` abstract class, and fill the `execute` method. The given `TaskResult... results` arguments permit to get the results from previous dependent tasks that have finished their execution.

As shown in the following lines, the given array of `TaskResults(results)` will be an array of two results (`TaskResult 2` and `3`) in this order if the dependences of `Task 5` is `Task 2` and `Task 3` in this order. Therefore you can use them to perform `Task 5` process.

```

@Override
public Object execute(TaskResult... results) throws Throwable {
    //TaskResult
    tResult2 = results[0];
    //TaskResult
    tResult3 = results[1];
}

```

Finally, overriding the `init()` method can be useful if you want to retrieve some parameters.

The task is the entity that will be scheduled by ProActive Scheduler. As it has been explained in the Section 1.3.2, "What is a Task ?", it's possible to create and add Java tasks to your TaskFlow Job. A Java task can also be created using an XML descriptor or the provided ProActive Scheduler Java API.

1.4.4.1.2. Create and Add a Java task using XML descriptor

Just take a look at the example below to understand the syntax of a task :

```
<task id="task1" retries="2">
<description>human description</description>
<javaExecutable class="org.ow2.proactive.scheduler.examples.WaitAndPrint">
  <parameters>
    <parameter name="foo" value="true"/>
    <parameter name="bar" value="1"/>
    <parameter name="test" value="toto"/>
  </parameters>
</javaExecutable>
</task>

<task id="task2">
  <depends>
    <task ref="task1"/>
  </depends>
  <javaExecutable class="org.ow2.proactive.scheduler.examples.WaitAndPrint">
    <parameters>
      <parameter name="foo" value="false"/>
      <parameter name="bar" value="12"/>
      <parameter name="test" value="titi"/>
    </parameters>
  </javaExecutable>
</task>
```

The Java Task is composed of one 'javaExecutable' that specified the 'executable' Java class to use. A set of parameters has also been defined to provide the executable some informations. These parameters will be available into the `HashMap` of the `init(HashMap)` method into your `JavaExecutable`. This example also shows the definition of two tasks with dependencies. We can easily see that 'task 2' depends on 'task 1'. So 'task 2' will be executed when 'task 1' has finished. To put these two tasks inside your `TaskFlow` job, just put it between the 'taskFlow' tags. Here's how a complete ready-to-be-scheduled `TaskFlow` Job seems like :

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:0.91"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:0.91
  http://proactive.inria.fr/schemas/jobdescriptor/0.91/schedulerjob.xsd"
  id="job_name" priority="normal" projectName="project_name" cancelOnError="true" logFile=
"path/to/a/log/file.log">
  <description>Job description</description>
  <variables>
    <variable name="val1" value="toto"/>
  </variables>
  <genericInformation>
    <info name="var1" value="{val1}"/>
    <info name="var2" value="val2"/>
  </genericInformation>

  <taskFlow>

    <task id="task1" retries="2">
      <description>human description</description>
      <javaExecutable class="org.ow2.proactive.scheduler.examples.WaitAndPrint">
```

```

<parameters>
  <parameter name="foo" value="true"/>
  <parameter name="bar" value="1"/>
  <parameter name="test" value="toto"/>
</parameters>
</javaExecutable>
</task>

<task id="task2">
  <depends>
    <task ref="task1"/>
  </depends>
  <javaExecutable class="org.ow2.proactive.scheduler.examples.WaitAndPrint">
    <parameters>
      <parameter name="foo" value="false"/>
      <parameter name="bar" value="12"/>
      <parameter name="test" value="titi"/>
    </parameters>
  </javaExecutable>
</task>

</taskFlow>
</job>

```

It is obviously possible to mix Java and Native task inside a taskFlow Job. Some other stuffs and options can be set onto a Java task, here's two examples of what can be done with task XML descriptors :

```

  <task id="taskName" preciousResult="true">
<description>Testing the pre and post scripts.</description>
<selection>
  <script type="static">
    <file path="${SCRIPT_DIR}/host_selection.js">
      <arguments>
        <argument value="${EXCLUSION_STRING}"/>
      </arguments>
    </file>
  </script>
</selection>
<pre>
  <script>
    <file path="${SCRIPT_DIR}/set.js"/>
  </script>
</pre>
<javaExecutable class="org.ow2.proactive.scheduler.examples.PropertyTask"/>
<post>
  <script>
    <file path="${SCRIPT_DIR}/unset.js"/>
  </script>
</post>
</task>

```

```

  <task id="PI_Computation" walltime="00:10" >
<genericInformation>

```

```

<info name="name1" value="val1"/>
</genericInformation>
<javaExecutable fork="true" class=
"org.objectweb.proactive.extensions.scheduler.examples.MonteCarlo" >
  <forkEnvironment javaHome="" jvmParameters="-d32" -/>
  <parameters>
    <parameter name="steps" value="20"/>
    <parameter name="iterations" value="100000000"/>
  </parameters>
</javaExecutable>
</task>

```

To have an exhaustive list of which options are available and what they are suppose to do, just go to the task explanation section at Section 1.4.4.6, “Tasks options and explanations”.

1.4.4.1.3. Create and Add a Java task using Java API

To **create a Java task** use the **JavaTask** class. In this type, you must specify the class you want to start with, by mentioning a Class of your executable. (To make your own executable see the proper section Section 1.4.4.1.1, “Define your own Java executable”). In addition, you can add arguments with which the task will be launched. These launching arguments will be given to the Java executable as a Map. Just take a look at the example below to see how to use the task creation Java API (see also Java DOcumentation of the Scheduler to learn more) :

```

//create a Java Task with the default constructor that we'll call -'aTask'
JavaTask aTask = new JavaTask();
//then, set the desired options -: (for example)
aTask.setName("task 1");
aTask.setDescription("This task will do something...");
aTask.addGenericInformation("key", "value");
aTask.setPreciousResult(true);
aTask.setRerunnable(2);
aTask.setRestartOnError(RestartMode.ELSEWHERE);
aTask.setResultPreview(UserDefinedResultPreview.class);
//add arguments (optional)
aTask.addArgument("foo", new Boolean(true));
aTask.addArgument("bar", new Integer(12));
aTask.addArgument("test", "test1");
//add executable class or instance
pat.setExecutableClassName("org.ow2.proactive.scheduler.examples.WaitAndPrint");

//SCRIPTS EXAMPLE
//If the script to use is in a file or URL
String[] args = new String("foo", "bar");
File scriptFile = new File("path/to/script_file");
//URL scriptURL = new URL("url/to/script_file");
Script script = new SimpleScript(scriptFile, args);
// Script script = new SimpleScript(scriptURL, args);
aTask.setPreScript(script);
//If the script to use is in a Java string for example
Script script = new SimpleScript("Script_content", "type_of_language");
//where type_of_language can be any language supported by the underlying JRE
aTask.setPreScript(script);

//same construction for the post script
aTask.setPostScript(script);

```

```
//same construction for the selection script
//the last parameter is still not used in the current implementation
SelectionScript selScript = new SelectionScript(script, true);
aTask.setSelectionScript(selScript);
```

To complete your job by adding the task inside the job, just add it as followed :

```
//add the task to the job
job.addTask(aTask);
```

Here's some other features than can be performed on tasks such as dependencies or wallTime :

```
//admitting task 2 and task 3 has been create just before
//we have to create task 5.
//create a new task
JavaTask task5 = new JavaTask();
//... (fill task5 as describe above)
//then specify dependencies by using the addDependence(Task) method
task5.addDependence(task2);
task5.addDependence(task3);
//or use the addDependencies(list<Task>) method as shown
//task5.addDependencies(new ArrayList<Task>(task2,task3));
```

```
//set this task as forked
aTask.setFork(true);
//or set a walltime
aTask.setWallTime(10000);
//you can also define a fork environment (for example)
ForkEnvironment env = new ForkEnvironment();
env.setJavaHome("Your/java/home/path");
env.setJVMParameters("-d12");
aTask.setForkEnvironment(env);
```

To have an exhaustive list of which options are available and what they are for, just go to Section 1.4.4.6, “Tasks options and explanations”.

1.4.4.2. Create and Add a native task

Note : It is only possible to add a native task in a TaskFlow Job.

To learn how to create a TaskFlow Job, just go to Section 1.4.2, “Create a TaskFlow job”. Once your TaskFlow Job created, you can add as many native tasks as needed to perform an application. A native task can be any native application such as programs, scripts, process, etc...

1.4.4.2.1. Create and Add a native task using XML descriptor

Just take a look at the example below to understand the syntax of a native task :

```
<!-- This native task example shows a native executable directly started as a
command. --->
<task id="task1_native" retries="2">
```

```

<description>Will display 10 dots every 1s</description>
<nativeExecutable>
  <!-- Consider that the ${VAR_NAME} has been defined in the job description as describe in
the job creation section --->
  <staticCommand
    value="${WORK_DIR}/native_exec">
    <arguments>
      <argument value="1"/>
    </arguments>
  </staticCommand>
</nativeExecutable>
</task>
<!-- This native task example shows a native executable started by a shell script. --->
<task id="task2_native">
  <description>Will display 10 dots every 2s</description>
  <depends>
    <task ref="task1_native"/>
  </depends>
  <nativeExecutable>
    <staticCommand
      value="${SCRIPT_DIR}/launcher.sh">
      <arguments>
        <argument value="${WORK_DIR}/native_exec"/>
        <argument value="2"/>
      </arguments>
    </staticCommand>
  </nativeExecutable>
</task>

```

The native Task is composed of one 'nativeExecutable' that specified the 'executable' process to use. A set of parameters has also be defined to provide the executable some informations. These parameters will be append to the command line starting by your native executable. This example also shows the definition of two tasks with dependencies. We can easily see that 'task2_native' depends on 'task1_native'. So 'task2_native' will be executed when 'task1_native' has finished. To put these two tasks inside your TaskFlow job, just put it between the 'taskFlow' tags. Here's how a complete ready-to-be-scheduled TaskFlow Job seems like :

```

<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:0.91"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:0.91
http://proactive.inria.fr/schemas/jobdescriptor/0.91/schedulerjob.xsd"
  id="job_name" priority="normal" projectName="project_name" cancelOnError="true" logFile=
"path/to/a/log/file.log">
  <description>Job description</description>
  <variables>
    <variable name="WORK_DIR" value="path/to/your/working/dir" -/>
    <variable name="SCRIPT_DIR" value="path/to/your/script/dir" -/>
  </variables>
  <genericInformation>
    <info name="var1" value="${WORK_DIR}"/>
    <info name="var2" value="val2"/>
  </genericInformation>

  <taskFlow>

    <task id="task1_native" retries="2">

```

```

<description>Will display 10 dots every 1s</description>
<nativeExecutable>
  <!-- Consider that the ${VAR_NAME} has been defined in the job description as describe
in the job creation section --->
  <staticCommand
    value="${WORK_DIR}/native_exec">
    <arguments>
      <argument value="1"/>
    </arguments>
  </staticCommand>
</nativeExecutable>
</task>
<task id="task2_native" retries="2">
  <description>Will display 10 dots every 1s</description>
  <nativeExecutable>
    <staticCommand
      value="${SCRIPT_DIR}/launcher.sh">
      <arguments>
        <argument value="${WORK_DIR}/native_exec"/>
        <argument value="1"/>
      </arguments>
    </staticCommand>
  </nativeExecutable>
</task>

</taskFlow>
</job>

```

It is obviously possible to mix Java and Native task inside a taskFlow Job. Some other stuffs and options can be set onto a native task, here's two examples of what can be done with task XML descriptors :

```

  <task id="taskName" preciousResult="true">
<description>Testing the pre and post scripts.</description>
<selection>
  <script type="static">
    <file path="${SCRIPT_DIR}/host_selection.js">
      <arguments>
        <argument value="${ECLUSION_STRING}"/>
      </arguments>
    </file>
  </script>
</selection>
<pre>
  <script>
    <file path="${SCRIPT_DIR}/set.js"/>
  </script>
</pre>
<nativeExecutable>
  <staticCommand
    value="${WORK_DIR}/native_exec">
    <arguments>
      <argument value="1"/>
    </arguments>
  </staticCommand>
</nativeExecutable>

```

```
<post>
  <script>
    <file path="${SCRIPT_DIR}/unset.js"/>
  </script>
</post>
</task>
```

```
    <task id="PI_Computation" walltime="00:10" >
<genericInformation>
  <info name="name1" value="val1"/>
</genericInformation>
<nativeExecutable>
  <!-- Consider that the ${VAR_NAME} has been defined in the job description as describe in
the job creation section --->
  <staticCommand
    value="${WORK_DIR}/native_exec">
    <arguments>
      <argument value="1"/>
    </arguments>
  </staticCommand>
</nativeExecutable>
</task>
```

To have an exhaustive list of which options are available and what they are suppose to do, just go to the task explanation section at Section 1.4.4.6, “Tasks options and explanations”.

1.4.4.2.2. Create and Add a native task using Java API

To **create a native task** use the **NativeTask** class. In this type, you must specify the executable you want to start, by mentioning a 'command line'. In addition, you can add arguments with which the task will be launched. These launching arguments will be append to the 'command line'. Just take a look at the example below to see how to use the task creation Java API (see also Java DOCumentation of the ProActive Scheduler to learn more) :

```
//create a native task with the default constructor that we'll call -'aTask'
NativeTask aTask = new NativeTask();
//then, set the desired options -: (for example)
aTask.setName("task 1");
aTask.setDescription("This task will do something...");
aTask.addGenericInformation("key", "value");
aTask.setPreciousResult(true);
aTask.setRerunnable(2);
aTask.setRestartOnError(RestartMode.ELSEWHERE);
aTask.setResultPreview(UserDefinedResultPreview.class);
//set the command line with the parameter append to the process name
aTask.setCommandLine("/path/to/command/cmd param1 param2");

//SCRIPTS EXAMPLE
//If the script to use is in a file or URL
String[] args = new String("foo", "bar");
File scriptFile = new File("path/to/script_file");
//URL scriptURL = new URL("url/to/script_file");
Script script = new SimpleScript(scriptFile, args);
```

```
// Script script = new SimpleScript(scriptURL, args);
aTask.setPreScript(script);
//If the script to use is in a Java string for example
Script script = new SimpleScript("Script_content", "type_of_language");
//where type_of_language can be any language supported by the underlying JRE
aTask.setPreScript(script);

//same construction for the post script
aTask.setPostScript(script);

//same construction for the selection script
//the last parameter is still not used in the current implementation
SelectionScript selScript = new SelectionScript(script, true);
aTask.setSelectionScript(selScript);
```

To complete your job by adding the task inside the job, just add it as followed :

```
//add the task to the job
job.addTask(aTask);
```

Here's some other features than can be performed on tasks such as dependencies or wallTime :

```
//admitting task 2 and task 3 has been create just before
//we have to create task 5.
//create a new task
NativeTask task5 = new NativeTask();
//... (fill task5 as describe above)
//then specify dependencies by using the addDependence(Task) method
task5.addDependence(task2);
task5.addDependence(task3);
//or use the addDependencies(list<Task>) method as shown
//task5.addDependencies(new ArrayList<Task>(task2,task3));
```

```
//set a walltime to stop the process after the given time even it is not finish
aTask.setWallTime(10000);
```

Here's a last example that describe how to create a native task with a **dynamic command**, i.e. generated by a script called a generation script. The generation script can only be associated to a **native** task: the execution of a generation script must set the string variable **command**. The value of this variable is the command line that will be executed by the ProActive Scheduler as task execution.

```
//create a new native task
NativeTask task2 = new NativeTask();
//create a generation script with a script as shown above
GenerationScript gscript = new GenerationScript(script);
//set the command to execute as a string
task2.setGenerationScript(gscript);
```

To have an exhaustive list of which options are available and what they are for, just go to Section 1.4.4.6, “Tasks options and explanations”.

1.4.4.3. Create and Add a ProActive task

Note : It is only possible to add a ProActive task only in a ProActive Job.

To learn how to create a ProActive Job, just go to Section 1.4.3, “Create a ProActive job”. Once your ProActive Job created, it is possible to just add ONE ProActive task inside your job.

1.4.4.3.1. Define your own ProActive executable

First of all, you must know that you can create your own ProActive executable by implementing scheduler executable interfaces. What is called 'executable' is in fact, the executed process (that is a Java class in this case). Here's an example to create your own ProActive executable application :

```
public class ProActiveExample extends ProActiveExecutable {

    private int numberToFind = 5003;

    @Override
    public Object execute(ArrayList<Node> nodes) {
        System.out.println("ProActive job started -!!");

        // create workers (on local node)
        Vector<Worker> workers = new Vector<Worker>();

        for (Node node -: nodes) {
            try {
                Worker w = (Worker)PAActiveObject.newActive(Worker.class.getName(),
                    new Object[] { -, }, node);
                workers.add(w);
            } catch (ActiveObjectCreationException e) {
                e.printStackTrace();
            } catch (NodeException e) {
                e.printStackTrace();
            }
        }

        // create controller Controller controller = new Controller(workers);
        int result = controller.findNthPrimeNumber(numberToFind);

        System.out.println("last prime -: -" + result);

        return result;
    }
}
```

As shown in a ProActive tutorial, this example uses the given nodes with the ProActive API in order to start 'workers' on them. The `execute(nodes)` method shows what can be done inside this kind of task. For more details about how to use the ProActive API, see the appropriate documentation. The complete example file can be found under 'jobs_descriptors/Job_ProActive.xml'.

1.4.4.4. Create and Add a ProActive task using XML descriptor

Just take a look at the example below to understand the syntax of the ProActive task :

```
<task id="Controller">
<description>Will control the workers in order to find the prime number</description>
```

```

<proActiveExecutable
  class="org.ow2.proactive.scheduler.examples.ProActiveExample">
  <parameters>
    <parameter name="numberToFind" value="200"/>
  </parameters>
</proActiveExecutable>
</task>

```

The ProActive Task is composed of one 'proActiveExecutable' that specified the 'ProActiveExecutable' Java class to use. A set of parameters has also be defined to provide this executable some informations. These parameters will be available into the `HashMap` of the `init(HashMap)` method into your `ProActiveExecutable`. To put this task inside your ProActive job, just put it between the 'ProActive' tags. On this job it is necessarily to set the number of node you desired for your `ProActiveExecutable`. Instead of deploying resources as it must be done in ProActive Suite, the resources are provides by the ProActive Scheduler. Here's how a complete ready-to-be-scheduled ProActive Job seems like :

```

  <?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:0.91"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:0.91
  http://proactive.inria.fr/schemas/jobdescriptor/0.91/schedulerjob.xsd"
  id="job_name" priority="normal" projectName="project_name" cancelOnError="true" logFile=
"path/to/a/log/file.log">
  <description>Job description</description>
  <variables>
    <variable name="WORK_DIR" value="path/to/your/working/dir" -/>
    <variable name="SCRIPT_DIR" value="path/to/your/script/dir" -/>
  </variables>
  <genericInformation>
    <info name="var1" value="{WORK_DIR}"/>
    <info name="var2" value="val2"/>
  </genericInformation>

  <proActive neededNodes="10">

    <task id="Controller">
      <description>Will control the workers in order to find the prime number</description>
      <proActiveExecutable
        class="org.ow2.proactive.scheduler.examples.ProActiveExample">
        <parameters>
          <parameter name="numberToFind" value="200"/>
        </parameters>
      </proActiveExecutable>
    </task>

  </proActive>
</job>

```

Some other stuffs and options can be set onto a ProActive task, here's a new example of what can be done with task XML descriptors :

```

  <task id="Controller">
    <description>Will control the workers in order to find the prime number</description>
    <selection>
      <script type="static">

```

```

<file path="${SCRIPT_DIR}/host_selection.js">
<arguments>
  <argument value="${EXCLUSION_STRING}"/>
</arguments>
</file>
</script>
</selection>
<pre>
<script>
  <file path="${SCRIPT_DIR}/set.js"/>
</script>
</pre>
<proActiveExecutable
  class="org.ow2.proactive.scheduler.examples.ProActiveExample">
  <parameters>
    <parameter name="numberToFind" value="200"/>
  </parameters>
</proActiveExecutable>
<post>
  <script>
    <file path="${SCRIPT_DIR}/unset.js"/>
  </script>
</post>
</task>

```

To have an exhaustive list of which options are available and what they are suppose to do, just go to the task explanation section at Section 1.4.4.6, “Tasks options and explanations”.

1.4.4.5. Create and Add a ProActive task using Java API

To **create a ProActive task** use the **ProActiveTask** class. In this type, you must specify the class you want to start with, by mentioning a Class of extending ProActiveExecutable. (To make your own executable see the proper section Section 1.4.4.3.1, “Define your own ProActive executable”). In addition, you can add arguments with which the task will be launched. These launching arguments will be given to the ProActive executable as a Map. Just take a look at the example below to see how to use the task creation Java API (see also Java DOCumentation of the Scheduler to learn more) :

```

//create a ProActive Task using the default constructor that we'll call -'aTask'
ProActiveTask aTask = new ProActiveTask();
//then, set the desired options -: (for example)
aTask.setName("task 1");
aTask.setDescription("This task will do something...");
aTask.addGenericInformation("key", "value");
aTask.setResultPreview(UserDefinedResultPreview.class);
//add arguments (optional)
aTask.addArgument("foo", new Boolean(true));
aTask.addArgument("bar", new Integer(12));
aTask.addArgument("test", "test1");
//add executable class or instance
pat.setExecutableClassName("org.ow2.proactive.scheduler.examples.ProActiveExample");
//add number of nodes needed for the application
pat.setNumberOfNodesNeeded(10);

//SCRIPTS EXAMPLE
//If the script to use is in a file or URL
String[] args = new String("foo", "bar");

```

```
File scriptFile = new File("path/to/script_file");
//URL scriptURL = new URL("url/to/script_file");
Script script = new SimpleScript(scriptFile, args);
// Script script = new SimpleScript(scriptURL, args);
aTask.setPreScript(script);
//If the script to use is in a Java string for example
Script script = new SimpleScript("Script_content", "type_of_language");
//where type_of_language can be any language supported by the underlying JRE
aTask.setPreScript(script);

//same construction for the post script
aTask.setPostScript(script);

//same construction for the selection script
//the last parameter is still not used in the current implementation
SelectionScript selScript = new SelectionScript(script, true);
aTask.setSelectionScript(selScript);
```

To complete your job by adding the task inside the job, just add it as followed : (note that you can only add ONE ProActive task in a ProActive Job)

```
//add the task to the job
job.addTask(aTask);
```

To have an exhaustive list of which options are available and what they are for, just go to Section 1.4.4.6, “Tasks options and explanations”.

1.4.4.6. Tasks options and explanations

As it has been shown in the different examples, it is possible to create 3 types of tasks. These 3 types have some common features like name, description, scripts, etc... Here's the details of each of these common features :

- **id** is the name assigned to the task. It can be whatever you want as a String. This name must be unique for each task.
- **description** (optional) is a human readable description of the task. It is for human use only. This field is optional but it is better to set it.
- **generic informations** (optional) is a way to define some informations inside your task. This informations could be read inside the policy (similar to job's one). It can be useful to add new complex scheduling behavior.
- **preciousResult** (optional - default is false) is the way to define that a result of a task is important or not. For example, in a job result, you could have to retrieve only some task results that are important for you. By setting the precious result to 'true', you'll be able to retrieve easily these results.
- **retries** (optional - default is 1) is a way to define how many times a task will be reran if a network problems occur. Set this value to **n** if you want the task to be restarted **n** times and so, started a maximum of **n+1** times.
- **restartOnError** (optional - default is false) is an option that define if a task has to be restarted if an error occurred. Error can be both exception for Java task or error code (1-255) for native task. It is a way to managed user error. If not defined, the task will never restart. This option can be set to **anywhere** that means the task will restart on the first available node. It can also be set to **elsewhere** meaning that the task will restart on a different node that the last used. In these 2 last cases, the job will be failed if the maximum number of retries (**retries** option) is reached. (This option is not available for proActive Task)

Combined with the job **cancelOnError** option it can be useful to know the behavior of your job. Here's a table that explains what can be done with tasks and job :

CancelOnError & RestartOnError mechanism							
CancelOnError		False			True		
RestartOnError		False	Anywhere	Elsewhere	False	Anywhere	Elsewhere
Native Task Result	-1*						
	0						
	1-255		**	**		**	**
	Exception		**	**		**	**
Java Task Result	Object						
	Exception		**	**		**	**

Legend :

	Task is restarted without condition
	Task result is returned at the end of the first execution
	Task is retried on the first ready node until the define number of retries (default is one)
	Task is retried on the first ready node - that is different from the previous one - until the define number of retries (default is one)
	Job will be canceled at the end of the first execution

* Used for internal management only

** Job will be failed at the end of retries

Figure 1.2. CancelOnError and RestartOnError behavior

- **Walltime** (optional) Task Walltime is a maximum allowed execution time of a task. It can be specified for any task, irrespectively of its type. If a task does not finish before its walltime it is terminated by the ProActive Scheduler. An example has been given above with the walltime specified. Note that, the walltime is defined in a task, thus it can be used for any type of a task. The general format of the walltime attribute is [hh:mm:ss], where h is hour, m is minute and s is second. The format still allows for more flexibility. We can define the walltime simply as “5” which corresponds to 5 seconds, “10” is 10 seconds, “4:10” is 4 minutes and 10 seconds, and so on. The walltime mechanism is started just before a task is launched. If a task does finish before its walltime, the mechanism is canceled. Otherwise, the task is terminated. Note that, the tasks are terminated without any prior notice. If the walltime is specified for a Java task (as in the example) it enforces the creation of a forked Java task instead.
- **fork and forkEnvironment** (optional only for Java Task) The purpose of a new type of a task Forked Java Task is to gain more flexibility with respect to the execution environment of a task. A new JVM is started with an inherited classpath and (possibly) redefined Java home path and JVM properties. It allows to use a JVM from a different provider and specify options to be passed to JVM (like memory usage). A Forked Java Task is defined as a Java Task with a forkEnvironment element. The aim of a forkEnvironment element is providing javaHome and jvmParameters attributes. For any undefined attribute a default environment value will be applied. Note that, the javaHome attribute points only to the Java installation directory and not the Java application itself. If the javaHome is not specified then the ProActive Scheduler will execute simply a Java command assuming that it is defined in the user path. The 'jvmParameters' attribute is a string composed of a sequence of Java options divided by a space.
- **parameters** (optional, only for Java and ProActive Task) is a way to define some parameters to be transfered to the executable. This is best explained in Section 1.4.4.1.1, “Define your own Java executable”. Each parameters is define with a name and a value and will be passed to the Java Executable as an HashMap.

arguments (optional, only for native Task) is a way to define arguments for your native process. Each arguments is define by a value that will be append to the process name to create a command line.

- **resultPreview** (optional) allows to specify how the result of a task should be displayed in the Scheduler graphical client. The user should implement a result preview class (that extends

`org.objectweb.proactive.extensions.scheduler.common.task.ResultPreview` (abstract class) which specifies result rendering in two different manners :

- a textual manner, by implementing `public abstract String getTextualDescription(TaskResult result);` . This method, similarly to `String Object.toString()` should return a `String` object that describes the result;
- a graphical manner, by implementing `public abstract JPanel getGraphicalDescription(TaskResult result);` . This method should return a `Swing JPanel` object that describes the result.

Some useful methods to create a specific preview class can be found in `org.objectweb.proactive.extensions.scheduler.common.task.util.ResultPreviewTool` , such as automatic display of an image file, or automatic translation between windows and unix path.

- **scripts** (optional) The ProActive scheduler supports portable scripts execution through the JSR 223 Java Scripting capabilities; scripts can be written in any language supported by the underlying Java Runtime Environment. Scripts are used in the ProActive scheduler to :
 - Execute some simple pre and post processing: optional pre-script and post-script
 - Select among available resources the node that suitable for the execution: optional selection-script can be associated to a task.
 - Dynamic building of a command line for a native task: optional generation-script (detailed in next section).

Here are some details and examples:

- **pre-script** The pre-script is always executed on the node that has been selected by the resource manager **before** the execution of the task itself.
- **post-script** The pre-script is always executed on the node that has been selected by the resource manager **after** the execution of the task itself.
- **selection script** The selection script is always executed before the task itself on any candidate node: the execution of a selection script must set the boolean variable `selected` , that indicates if the candidate node is suitable for the execution of the associated task.

Now that you have your job created, next step is to submit it to the ProActive Scheduler.

1.4.5. Submit a job to the ProActive Scheduler

The submission will perform some checking to ensure that a job is correctly formed. Then the job is inserted in the pending list and wait for executions until free resources become available. Once done, the job will be started on the resources deployed by the resource manager. Finally, once finished, the job goes in a finish queue and will wait until user to retrieve its result. There are three ways to submit a job to The Scheduler :

1.4.5.1. Submit a job using the Graphical User Interface (Scheduler Eclipse Plugin)

To submit a job using the graphical tools, you must have first created a job XML Descriptor. Then refer to Chapter 2, *ProActive Scheduler Eclipse Plugin* documentation to submit it.

1.4.5.2. Submit a job using shell command

Use the provided shell script `jobLauncher.sh` to submit a job using command line. This script (`bin/unix/jobLauncher.sh`) has 1 mandatory option and 3 optional :

- **The path to the job** file descriptor is mandatory (using the `"-j PATH"` option)
- **The URL of a started scheduler.** (using the `"-u URL"` option) If not mentioned, the script will connect an existing localhost Scheduler.
- **Your login** (using the `"-l LOGIN"` option). If you use this option, only your password will be requested. Otherwise, both will be.
- **The number of jobs** to submit, by default only 1 will be submitted (using the `"-n A_NUMBER"` option).

For example : `./jobLauncher.sh -j ../jobs_descriptors/Job_with_dep.xml -l login -n 12 -u //localhost/` will submit 12 times the `Job_with_dep` job to a local ProActive Scheduler and only your password will be requested. Authorized username and password are defined by the administrator.

For more informations, use `-h` (or `--help`) option (i.e. `"jobLauncher.sh -h"`)

1.4.5.3. Submit a job using Java API

To connect the ProActive Scheduler and submit a Job using Java API, just proceed as following :

```
//join an existing ProActive Scheduler retrieving an authentication interface.
SchedulerAuthenticationInterface auth = SchedulerConnection.join("//host/SCHEDULER_OBJECT_NAME");
//connect and log to the Scheduler. Valid username and password are define by the administrator.
UserSchedulerInterface scheduler = auth.logAsUser("username", "password");
// submitting a new job and get the associated id
JobId myJobId = scheduler.submit(job);
```

As you can see submitting a job will return a Job ID. This is the identification code of the submitted Job. It is useful to save it in order to retrieve future informations on this job.

1.4.6. Get a Job result

Once a Job terminated, it is possible to get its result. You can only get the result of the job that you own.

1.4.6.1. Get a Job result using the Graphical User Interface (Scheduler Eclipse Plugin)

To get a job result using the graphical tools, please refer to Chapter 2, *ProActive Scheduler Eclipse Plugin* documentation.

1.4.6.2. Get a Job result using shell command

To get the result of a job using a command line, use the **getResult.sh** script in the (bin/unix/getResult.sh) directory. This script has 2 optional options :

- The URL of a started scheduler. (using the "-u URL" option). If you don't use this, it will try to connect to a started scheduler on local host.
- Your login (using the "-l LOGIN" option). If you use this option, only your password will be requested. Otherwise, both will be requested.

It will print the result on the screen as the toString() Java method could have done it.

For more informations, use -h (or --help) option (i.e. "jobLauncher.sh -h")

1.4.6.3. Get a Job result using Java API

To do it in Java, use the **getJobResult(JobId)** method in the **UserSchedulerInterface** and the job ID you got when you submitted it. It is also possible to create a new ID based on the integer id you got. A job result is in fact a list of task result ordered in three lists :

- A full list that contains every result or exception of every tasks.
- A failed list that contains every result or exception returned by a task that failed.
- And a precious result list that contains every result or exception returned by the task marked precious.

This result will be given to you exactly like you returned it in your executable. To know when a job that you have submitted has finished its execution, you can subscribe to the scheduler to be notified of some events. This will be explain in the next section.

```
// get the user interface
UserSchedulerInterface scheduler = auth.logAsUser("username", "password");
// get the result of the job
JobResult myResult = scheduler.getJobResult(myJobId);
//look at inside the JobResult to retrieve TaskResult...
```

1.4.7. Register to ProActive Scheduler events

If you are **using the Java API**, it is possible to get events from the Scheduler. In order to be notified about the scheduler activities, you can add a Scheduler listener that will inform you of some events, like job submitting, job or task finished, scheduling state changing, etc... To add a listener, just make your listener by implementing the **SchedulerEventListener** interface and add it to the scheduler. You will then receive the scheduler initial state containing some informations about the current scheduling state. See the ProActive Scheduler JAVADOC for more details.


```
//make your listener
SchedulerEventListener mySchedulerEventListener = new SchedulerEventListener () {
    public void jobRunningToFinishedEvent(JobEvent event){
        //if my job is finished
        if (event.getJobId().equals(myJobId)){
            //get its result
            JobResult myResult = scheduler.getJobResult(myJobId);
        }
    }
    //Implement other methods...
}
//add the listener to the scheduler specified which events you want to receive.
scheduler.addSchedulerEventListener(MySchedulerEventListener, SchedulerEvent.JOB_RUNNING_TO_FINISHED);
```

This example shows you how to listen to the scheduler events (here the finished job event only). But you can listen for every events you want containing in this interface.

For more details and features on the user scheduler interface, please refer to the java Documentation.

1.5. Administrator Manual

TODO

Chapter 2. ProActive Scheduler Eclipse Plugin

The **Scheduler Eclipse Plugin** is a **graphical client** for remote monitoring and control of the ProActive Scheduler (see Chapter 1, *ProActive Scheduler*), including remote submission of XML-defined jobs (see Section 1.4.1.1, “Create a job using XML descriptor”).

The Scheduler Eclipse Plugin is available in two forms :

- A **Java stand alone application** based on Eclipse Rich Client Platform (RCP) [http://wiki.eclipse.org/index.php/Rich_Client_Platform] , available for any platform (Windows, Linux, Mac OSX, Solaris, ...)
- A set of **Eclipse** [<http://www.eclipse.org>] **plugins** : with all the functionalities within the stand alone application, enhanced with a tool that makes easier the scheduling and monitoring of jobs and applications.

2.1. The Scheduler perspective

The Scheduler plugin provides the **Scheduler perspective** [<http://help.eclipse.org/help31/index.jsp?topic=/org.eclipse.platform.doc.user/gettingStarted/qs-43.htm>] displayed in the Figure 2.1, “The Scheduler Perspective” .

This perspective defines the following set of views [http://wiki.eclipse.org/index.php/FAQ_What_is_a_view%3F] :

- The **Jobs** view: shows pending, running and finished jobs in the scheduler.
- The **Console** view: shows jobs standard and error output (on demand).
- The **Tasks** view: displays detailed informations on tasks contained in the selected job.
- The **Jobs Info** view: displays all informations of the selected job.
- The **Result Preview** view: displays a textual or graphical preview of the result of the selected task.

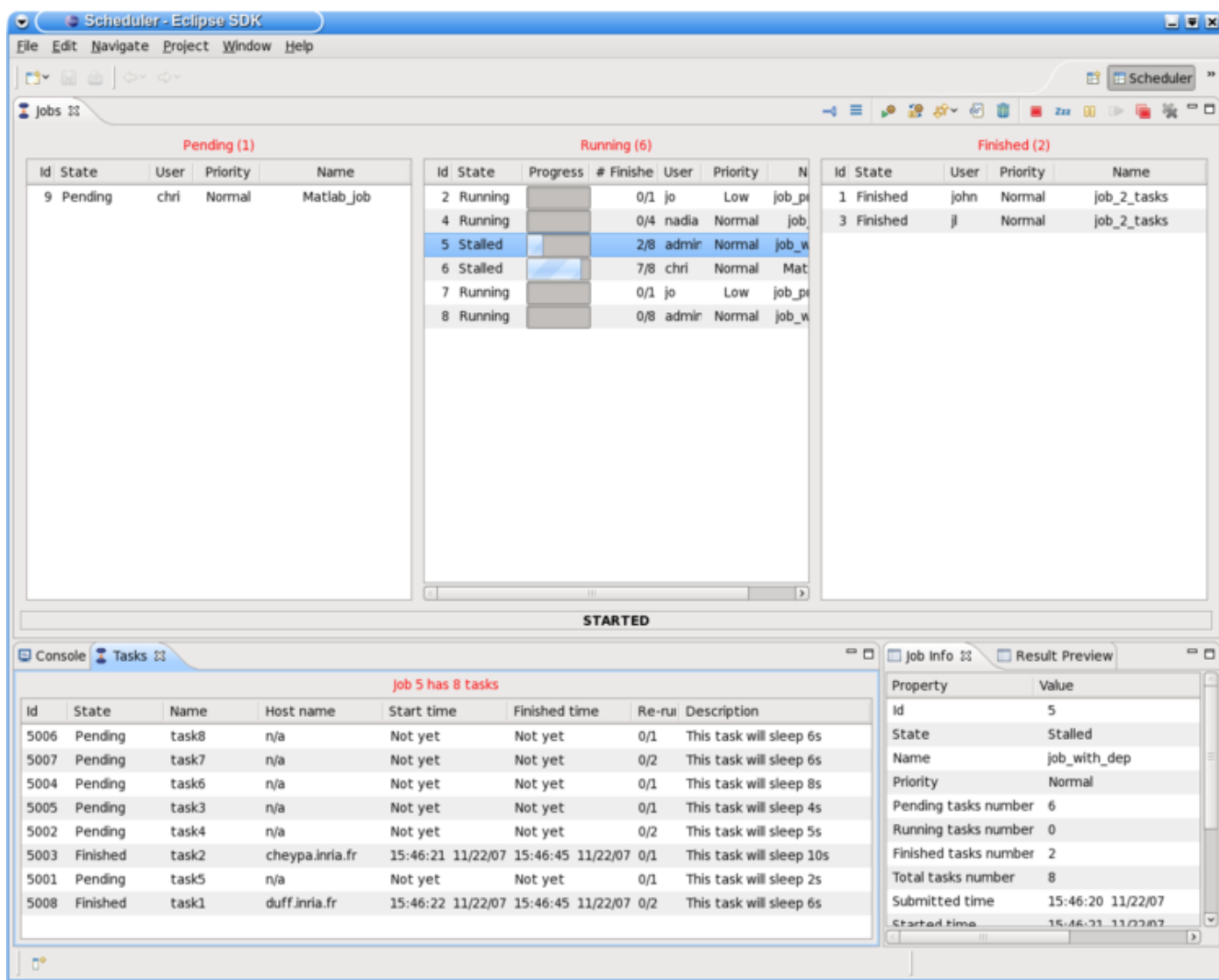


Figure 2.1. The Scheduler Perspective

2.2. Views composing the perspective

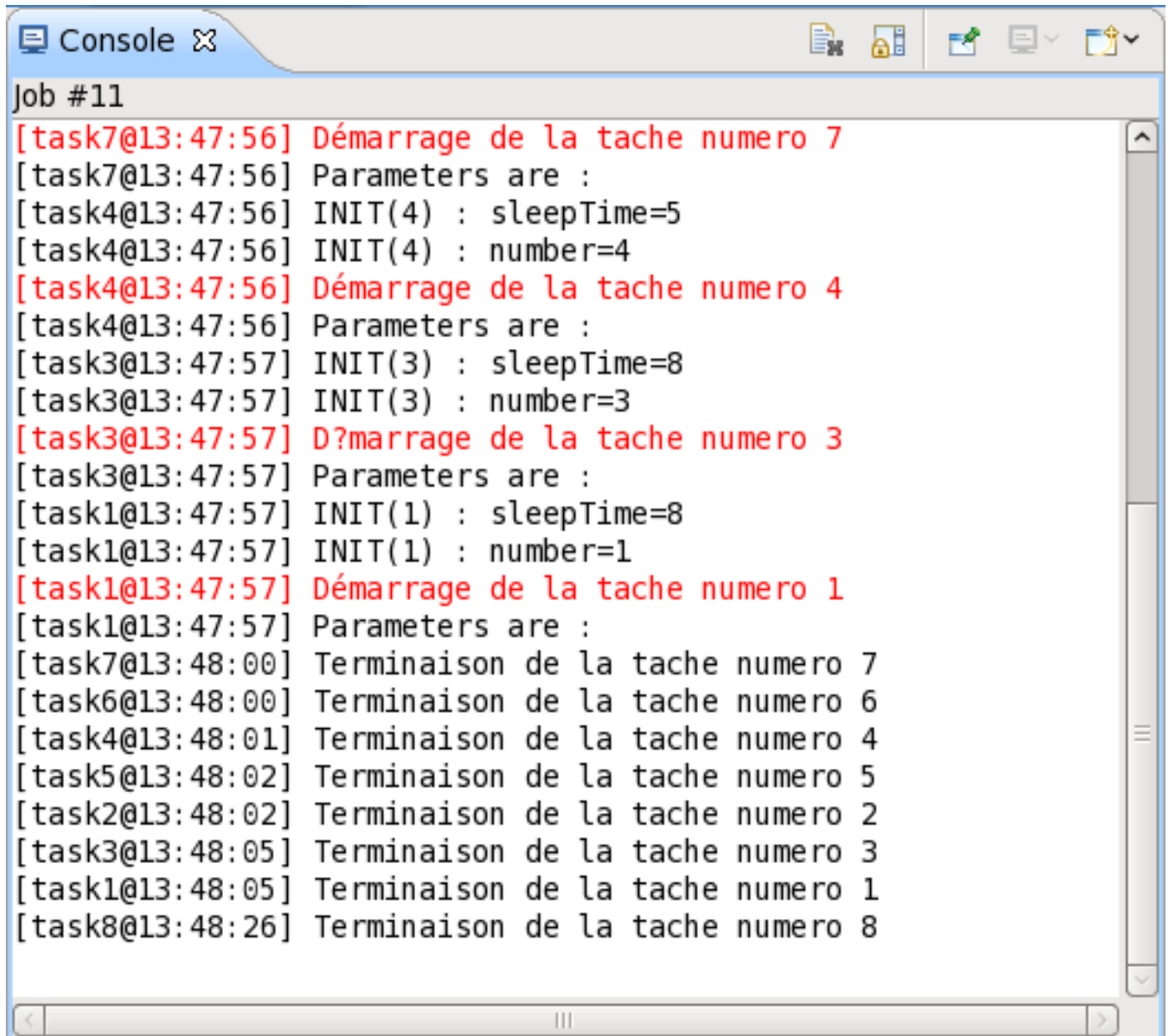
Pending (7)					Running (11)					Finished (8)				
Id	State	User	Priority	Name	Id	State	Progress	# Finish	Use	Id	State	User	Priority	Name
362	Pending	chri	Low	job_proAc	351	Running	<div><div></div></div>	3/4	jl	49	Finished	chri	Normal	job_f
375	Pending	admin	Normal	Job_with_	361	Running	<div><div></div></div>	7/8	jl	50	Finished	chri	Normal	job_f
376	Pending	jo	Normal	Job_with_	364	Running	<div><div></div></div>	0/8	jo	354	Finished	admin	Normal	Matlab
377	Pending	jo	Normal	Job_with_	366	Running	<div><div></div></div>	1/4	nad	350	Finished	jl	Normal	job_na
379	Pending	chri	Normal	job_P	367	Running	<div><div></div></div>	1/4	nad	363	Finished	jo	Normal	Matlab
380	Pending	jo	Low	job_proAc	368	Running	<div><div></div></div>	1/9	johr	365	Finished	admin	Normal	job_pre
381	Pending	jo	Low	job_proAc	369	Running	<div><div></div></div>	0/9	johr	370	Finished	chri	Normal	job_pre
					371	Running	<div><div></div></div>	2/8	jl	372	Finished	admin	Normal	job_pre
					373	Running	<div><div></div></div>	1/4	nad					
					374	Running	<div><div></div></div>	2/8	adm					
					378	Running	<div><div></div></div>	0/8	chri					

RESUMED

Figure 2.2. The Jobs view

All buttons (on upper right) are describe in the

This view is composed of 3 tables which represents pending, running and finished jobs. In each table you can watch many different informations about jobs, as their state, their name, their id...



```
Job #11
[task7@13:47:56] Démarrage de la tache numero 7
[task7@13:47:56] Parameters are :
[task4@13:47:56] INIT(4) : sleepTime=5
[task4@13:47:56] INIT(4) : number=4
[task4@13:47:56] Démarrage de la tache numero 4
[task4@13:47:56] Parameters are :
[task3@13:47:57] INIT(3) : sleepTime=8
[task3@13:47:57] INIT(3) : number=3
[task3@13:47:57] Démarrage de la tache numero 3
[task3@13:47:57] Parameters are :
[task1@13:47:57] INIT(1) : sleepTime=8
[task1@13:47:57] INIT(1) : number=1
[task1@13:47:57] Démarrage de la tache numero 1
[task1@13:47:57] Parameters are :
[task7@13:48:00] Terminaison de la tache numero 7
[task6@13:48:00] Terminaison de la tache numero 6
[task4@13:48:01] Terminaison de la tache numero 4
[task5@13:48:02] Terminaison de la tache numero 5
[task2@13:48:02] Terminaison de la tache numero 2
[task3@13:48:05] Terminaison de la tache numero 3
[task1@13:48:05] Terminaison de la tache numero 1
[task8@13:48:26] Terminaison de la tache numero 8
```

Figure 2.3. The Console view

This view displays all jobs standard and error output (only on demand).

Tasks 

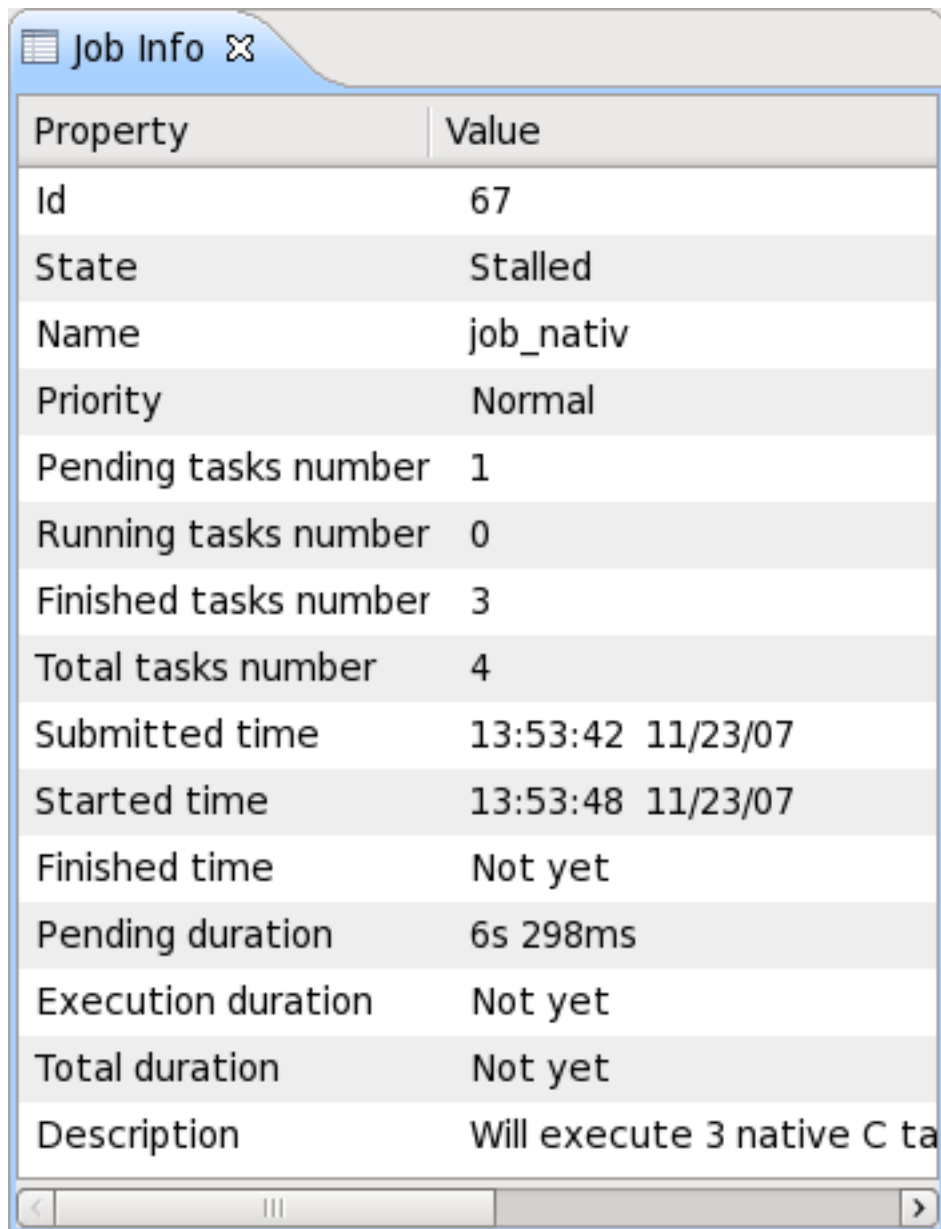
Job 50 has 9 tasks

Id	State	Name	Host name	Start time	Finished time	Re-run	Description
50007	Pending	Average1	n/a	Not yet	Not yet	0/1	Do the average of 1 2 3 and return it.
50005	Pending	LastAverage	n/a	Not yet	Not yet	0/1	Do the average of average 1 2 and return it.
50003	Pending	Average2	n/a	Not yet	Not yet	0/1	Do the average of 4 5 6 and return it.
50001	Running	Computation6	nahuel.inria.fr	13:52:39 11/23/07	Not yet	0/1	Compute Pi and return it
50008	Finished	Computation2	amda.inria.fr	13:52:36 11/23/07	13:53:14 11/23/07	0/1	Compute Pi and return it
50009	Finished	Computation5	puravida.inria.fr	13:52:24 11/23/07	13:53:09 11/23/07	0/1	Compute Pi and return it
50006	Finished	Computation3	pincoya.inria.fr	13:52:36 11/23/07	13:53:08 11/23/07	0/1	Compute Pi and return it
50004	Finished	Computation1	macyavel.inria.fr	13:52:37 11/23/07	13:53:09 11/23/07	0/1	Compute Pi and return it
50002	Finished	Computation4	trans04.inria.fr	13:52:38 11/23/07	13:53:09 11/23/07	0/1	Compute Pi and return it

Figure 2.4. The Tasks view

This view provides many informations on tasks composing a job as :

- The task id
- The task state
- The task name
- The host name which execute the task
- The task started time
- The task finished time
- In the column "Re-runnable", the first number represents how many times the task was re-executed, and the second number how many times the task can be re-executed
- The description of the task



Property	Value
Id	67
State	Stalled
Name	job_nativ
Priority	Normal
Pending tasks number	1
Running tasks number	0
Finished tasks number	3
Total tasks number	4
Submitted time	13:53:42 11/23/07
Started time	13:53:48 11/23/07
Finished time	Not yet
Pending duration	6s 298ms
Execution duration	Not yet
Total duration	Not yet
Description	Will execute 3 native C ta

Figure 2.5. The Job Info view

This view provides many informations on the selected job as :

- The job id
- The job state
- The job name
- The job priority
- The number of pending task
- The number of running task
- The number of finished task
- The number of task composing the job
- The job submitted time
- The job started time
- The job finished time
- The description of the job

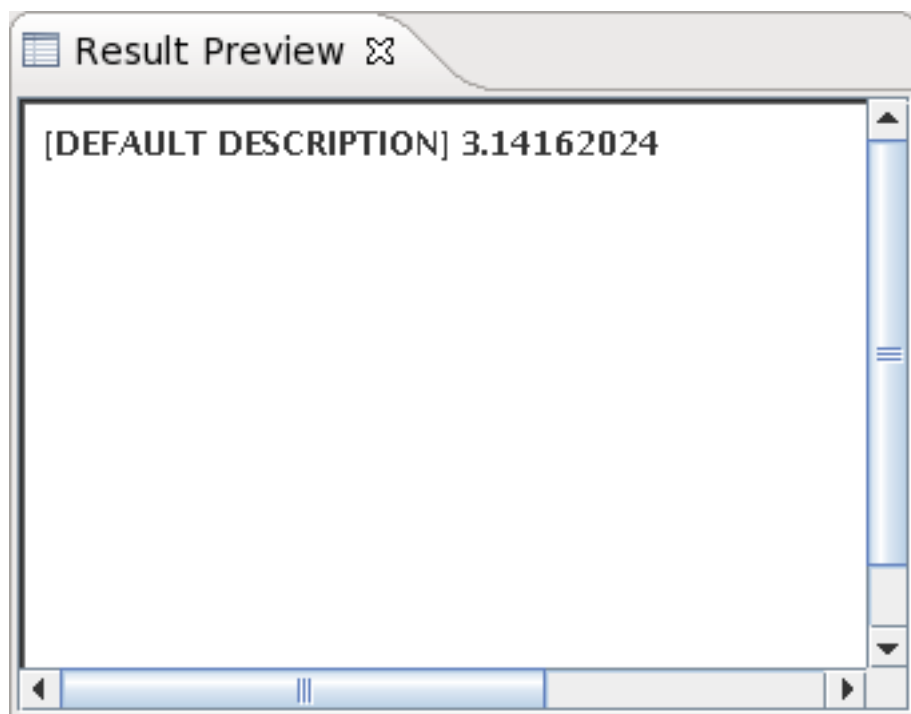


Figure 2.6. The Result Preview view

This view displays the result of the selected task (in task view), according to the ResultPreview field (see Section 1.4.4, “Create and Add a task to a job”)

2.3. Connect to the started ProActive Scheduler

In order to establish a connection to the ProActive Scheduler:

1. open the Scheduler Perspective: **Window->Open Perspective->Other...->Scheduler** (it could be already opened as it is the default perspective).
2. select **"Connect to the ProActive scheduler"** in the Scheduler menu or in the contextual menu (right click), it opens the "Connect to the ProActive scheduler" dialog displayed in the Figure 2.7, “Connect to scheduler”.
3. **enter informations required** about the remote scheduler, and click **OK**.

note: If you check "log as admin" in the previous dialogue, if the ProActive scheduler accepts your connection, you'll be able to do more actions than an "simple" user (see Section 2.4.2, “The Jobs view buttons in Administrator Mode”).

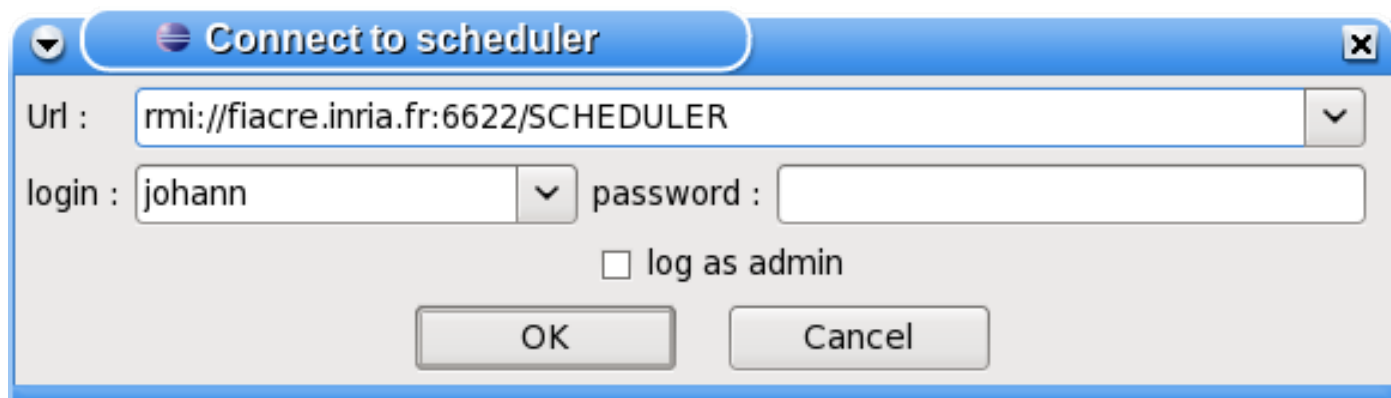


Figure 2.7. Connect to scheduler

2.4. The Scheduler perspective buttons

2.4.1. The Jobs view buttons in User Mode



Figure 2.8. Connect to scheduler

Display the "Connect to scheduler" dialog in order to establish a connection to a remote ProActive scheduler.



Figure 2.9. Disconnect from the scheduler



Figure 2.10. Change view from Vertical to Horizontal mode

Allows to switch the job's display to Horizontal from Vertical mode.



Figure 2.11. Change view from Horizontal to Vertical mode

Allows to switch the job's display from Horizontal to Vertical mode.



Figure 2.12. Submit a job

Display the "Choose file" dialog in order to submit a XML-defined job to the scheduler.



Figure 2.13. Pause/Resume a job

Pause or resume the selected job (only if you are the job owner).



Figure 2.14. Change job priority

Change job priority (only if you are the job owner). Priority allowed are:

- Lowest.
- Low.
- Normal.



Figure 2.15. Display job output

To display the selected job's standard and error output (only if you are the job owner).

**Figure 2.16. Kill Job**

To kill the selected job (only if you are the job owner).

2.4.2. The Jobs view buttons in Administrator Mode

All buttons allowed in user mode are also allowed in Administrator mode. Moreover you can execute any action even you aren't the job owner.

In Administrator mode, 3 other choices for job priority are available :

- Idle, lower priority than the 3 User Mode priorities
- High, higher priority than the 3 User Mode priorities
- Highest, higher priority than the 3 User Mode priorities and than High

**Figure 2.17. Start the scheduler****Figure 2.18. Stop the scheduler****Figure 2.19. Freeze the scheduler**

This freezes the scheduler. When the scheduler receives this event, it pauses all running jobs, and no other pending jobs will be scheduled.

**Figure 2.20. Pause the scheduler**

This pauses the scheduler. When the scheduler receives this event, no pending jobs will be scheduled, but all running jobs complete.

**Figure 2.21. Resume the scheduler****Figure 2.22. Shutdown the scheduler**

This shutdowns the scheduler. When the scheduler receives this event, job submission is no more allowed, but all running jobs complete. When all jobs are finished, the scheduler is shutdown.

**Figure 2.23. Kill scheduler**

This shutdown immediately the scheduler, without waiting for any job completion.

Part II. ProActive Resource Manager

Table of Contents

- Chapter 3. ProActive Resource Manager 36**
 - 3.1. IMPORTANT NOTE 36
 - 3.2. Role 36
 - 3.3. Resource Manager architecture 36
 - 3.4. Static Node Source and Dynamic Node Source 37
 - 3.5. Nodes states 37
 - 3.6. Starting the Resource Manager 38
- Chapter 4. Resource Manager's Eclipse Plugin 39**

Chapter 3. ProActive Resource Manager

3.1. IMPORTANT NOTE

- Some parts of the ProActive Scheduler and ProActive Resource Manager rely on Java Scripting capabilities (JSR 223 [<http://jcp.org/en/jsr/detail?id=223>]). As a consequence, it requires either:

- a 1.6 or greater Java Runtime Environment, without any modifications,
- or, with a 1.5 JRE, the JSR 223 jar files [<http://jcp.org/aboutJava/communityprocess/final/jsr223/index.html>] :
 - First, the `script-api.jar`, `script-js.jar` and `js.jar` files must be added in the `/ProActive/dist/lib/` directory if you are using the bin release or ProActive, or in the `/ProActive/lib/` directory if you build ProActive from the source release.
 - Then the `java5_jsr223_patch.jar` patch (released with the Scheduler RCP Client) should be executed in the Scheduler RCP Client directory : unzip the `java5_jsr223_patch.zip` file and execute `java -jar java5_jsr223_patch.jar` .

NOTE - A RCP/Eclipse graphical client is available, **but not yet documented** for monitoring and controlling a Resource Manager. You can find it in the ProActive Download page.

3.2. Role

As Scheduler manages pool of jobs to execute, Resource manager is in charge of supplying Scheduler in resources : ProActive nodes. Resource Manager (RM) takes benefits of the ProActive library, so it can handle resources from LAN, on cluster, on P2P desktop Grids, or on Internet Grids. ResourceManager provides scheduler in nodes, according to criteria of the task to execute on it (operating system, dynamic libraries, memory...). Its main functions are :

- Creation, acquisition and removal of ProActive nodes.
- Supplying nodes to scheduler for tasks executions, Scheduler can ask nodes that verify criteria, these criteria are defined in a selection script.
- Maintaining and monitoring its list of node resources, and manage states of its handled nodes (free, busy, down...).

3.3. Resource Manager architecture

Resource Manager is made of five components :

- **User** Resource Manager's frontend for the Scheduler, this component provides for scheduler an entry to get and give back nodes.
- **Admin** Frontend for RM's administrator, provides administrator actions; add and remove nodes, add and remove different node sources, shutting down the Resource manager.
- **Core** Main component, selects and gives nodes to scheduler, maintains different nodes states for each node, and receive new available nodes acquired by Node Sources.
- **Monitoring component** Resource manager can have monitors connected to it. Monitors are external programs (such as monitor GUI) that want to be informed about RM current activity; numbers of nodes and their availability for example. The Monitoring component is in charge of throwing RM information to its monitors.
- **Node Sources** Resource manager can handle nodes coming from heterogeneous environments, a Node source component is in charge of nodes acquisition, deployment and monitoring for a dedicated infrastructure. It means we can have a Node source which manages nodes deployed by a ProActive descriptor, one for nodes acquired from a Peer to peer infrastructure, and another for nodes acquired from a cluster.

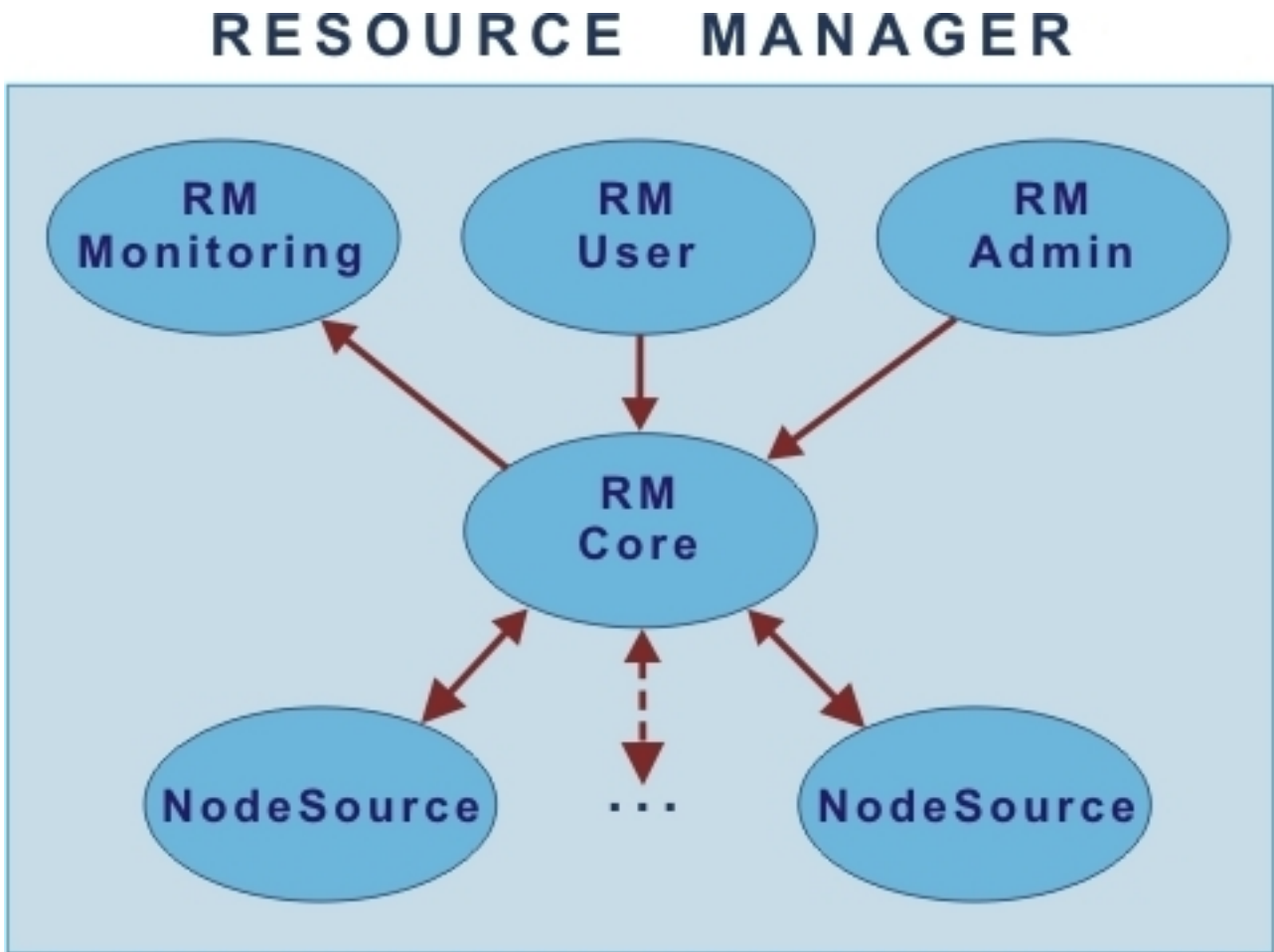


Figure 3.1. resource Manager architecture

3.4. Static Node Source and Dynamic Node Source

Node Sources objects are aimed to acquire nodes for the RM, there are two kinds of Node Sources :

- **Static Node Source** defined for deploying and acquiring nodes by a ProActive descriptor. All nodes handled by the source are kept permanently by the Resource Manager, i.e until the termination of the RM or if Administrator ask to remove some of them.
- **Dynamic Node Source** designed to acquire nodes from an infrastructure which can provide nodes just for a limited time. So this Node Source get a node from a specific infrastructure, keep it during a defined time. When this keeping time is elapsed, dynamic node source remove the node from the RM and give back node to its infrastructure. A dynamic node source have three main parameters :

Max number of nodes : number of nodes that dynamic Node Source has to get from its corresponding infrastructure. The dynamic node source tries to acquire this number of node source, but its infrastructure may not be able to provide as many nodes. So this is the number of acquired nodes that Dynamic Node Source tries to reach.

Time to release(TTR) : keeping duration of an acquired node. when this keeping duration is reached by a node, dynamic node source releases it.

Nice time : After a node release, time to wait for the dynamic node source before trying to get a new node from its infrastructure. After each node release, dynamic node source waits "nice time", and after tries to acquire a new node.

3.5. Nodes states

Resource Manager has to maintain states of its handled nodes, here the different nodes states :

- **Free** Node is available, and there is no task launched on it. Node can be supplied to a scheduler.

- **Busy** Node has been given to scheduler and a task is executed on it.
- **To be released** Node is busy, and administrator or its (dynamic) Node Source has asked to remove the node. So the node will be removed from RM after task's end.
- **Down** Node has a problem (unreachable, fallen...) and can't execute tasks anymore.

3.6. Starting the Resource Manager

To start the Resource Manager, run the `RMLauncher.sh` script in `scripts/scheduler` directory. Without arguments, Resource Manager will start and create four ProActive nodes on the local host. `RMLauncher.sh` can be started with 1 optional argument:

- Path of a ProActive descriptor file (for example : `ProActive/descriptors/Workers.xml`). Descriptor is deployed and nodes added to the RM by a static Node Source at the RM's startup.

You can also start Resource Manager using the java API. Resource Manager can be started with static functions of `RMFactory` class. Here a short sample of RM instantiation with deployment of a ProActive descriptor. Nodes deployment is asked to `RMAdmin` object :

```
RMFactory.startLocal(); //creates Resource Manager  
components RMAdmin admin = RMFactory.getAdmin(); //get  
RMAdmin object  
  
//creates ProActive Descriptor object from an xml file  
ProActiveDescriptor pad =  
PADeployment.getProactiveDescriptor("myDescriptor.xml");  
  
//Ask to RMAdmin component to deploy the ProActive  
Descriptor //and add deployed nodes to the RM  
admin.addNodes(pad);
```

Chapter 4. Resource Manager's Eclipse Plugin

Part III. ProActive Scheduler's Matlab extension

Table of Contents

Chapter 5. ProActive Scheduler's Matlab Extension	41
5.1. Presentation	41
5.2. Quick Start with the Matlab Extension	41
5.2.1. Installation	41
5.2.2. Writing a simple example : the Matlab Script	41
5.2.3. Writing a simple example : the Scheduler job descriptor	41
5.3. A More Complex Example : a Matlab task flow	45
5.3.1. Descriptor variables	47
5.3.2. New Tasks : MatlabSplitter and MatlabCollector	47
5.3.3. Task dependencies	47
5.3.4. New parameter in SimpleMatlab tasks: index	47
5.3.5. Matlab Scripts for this example	48

Chapter 5. ProActive Scheduler's Matlab Extension

5.1. Presentation

MATLAB is a numerical computing environment and programming language. Created by The MathWorks, MATLAB allows easy matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages.

The Goal of ProActive Scheduler's Matlab Extension is to:

- allow users to easily launch Matlab scripts over an heterogeneous grid
- describe complex task flows in Matlab using human-readable XML descriptors
- Ability to communicate the result of one task as input of another task.
- users won't have to write any other code than Matlab script code
- support the following Matlab types : Double, Complex, Int or String Arrays. Cells. Records.

A good way to start manipulating and scheduling Matlab scripts is to have a look at the standalone (and simple) Matlab/Scilab GUI described in the ProActive documentation. If you want to directly through more complex Matlab job scheduling, go on with the following tutorial.

5.2. Quick Start with the Matlab Extension

To get quickly our hands in, we'll write a very simple Matlab job example. This simple example will compute the roots of several polynomials.

5.2.1. Installation

Before starting to use the Matlab interface, you need to install the Matlab interface to Java. You'll find all the instructions on PROACTIVE/scripts/unix/matlab/README. This interface will build the native libraries of the Java Interface to Matlab. As this library is native, it is important that you build it for each couple <Matlab version, Architecture> inside your grid infrastructure. If you are using a ProActive installation on a centralized NFS folder, this will be sufficient. Otherwise, you will have to build and install the native library inside your ProActive installation on each machine used.

The good news are, if you successfully run the configuration script, you won't have to bother where Matlab is installed at runtime, the Scheduler will determine it for you. A little drawback to this is that the scheduler will use the first Matlab installation found on the system, so it might not do what you want when several Matlab installations are on the same machine. Further releases of the extension will allow a finer control over that by specifying Matlab's minimum version requirement inside job descriptors.

5.2.2. Writing a simple example : the Matlab Script

We write a very simple script which computes the roots of a single given polynomial.

```
out=roots(in);
```

The **in** and **out** variables are specific variables which describe the inputs and outputs of a Matlab script for the Scheduler. in and out can contain anything supported by the extension (Double, Complex, or String arrays, Cells, Records).

5.2.3. Writing a simple example : the Scheduler job descriptor

This is a step by step guide to write this job descriptor.

5.2.3.1. The job definition

The **job** tag is the root tag of our descriptor, it must have a **name** attribute which holds an id of the job. It is generally followed by a **description** tag which gives textual description of the job. Finally, the next tag will be the type of job to schedule. In our case it will be a **taskFlow** job (a job containing several tasks).

```
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:0.91
  http://proactive.inria.fr/schemas/jobdescriptor/0.91/schedulerjob.xsd"
  xmlns="urn:proactive:jobdescriptor:0.91" name="Matlab_job_simplest">
  <description>A simple Matlab job, which computes the roots of several polynomials</
description>
  <taskFlow>
    ~...
  </taskFlow>
</job>
```

5.2.3.2. The task definition

The **task** tag contains all the information for a single task executed on a single machine. In our example, this task will be the matlab script calculating the root of a polynomial.

The task tag must contain a **name** attribute like the job tag. Here it contains as well the attribute **preciousResult** which tells the scheduler that we need the result of this task as final output for our job. The task tag is immediately followed by a **description** tag containing a textual description of this task.

The description tag is followed by a **selection** tag. This tag describes a script which will select, among all the machine resources that the Scheduler controls, the specific resources (machine) that can effectively run this task. This script can for example test that Matlab is installed and has the right version, that specific Toolboxes are installed... We provide a generic script which simply tests if Matlab is installed. The script is retrieved from the URL <http://proactive.inria.fr/userfiles/file/scripts/checkMatlab.js>

```
<task name="root1" preciousResult="true">
  <description>Calculates the root of a polynomial</description>
  <selection>
    <script>
      <file url="http://proactive.inria.fr/userfiles/file/scripts/checkMatlab.js"/>
    </script>
  </selection>
  ~...
</task>
```

5.2.3.3. The Matlab script definition

Now we finally write the script that will be executed on the remote machine. The **javaExecutable** tag is a container for our Matlab script, it's a java program that will connect to the Matlab engine and launch the given script inside it.

```
<javaExecutable class="org.objectweb.proactive.extensions.scheduler.ext.matlab.SimpleMatlab">
  <parameters>
    <parameter name="input" value="in=[1 0 3 --2 5 1];"/>
    <parameter name="script" value="out=roots(in);"/>
  </parameters>
</javaExecutable>
```

The javaExecutable tag contains an attribute **class** which tells which type of Matlab task will be used, here we'll describe only the task called **SimpleMatlab**. In Section 5.3.2, “New Tasks : MatlabSplitter and MatlabCollector”, we describe more advanced tasks. The javaExecutable tag contains a child tag called **parameters**. This tag contains a list of **parameter** tags which define the task parameters. Each parameter tag, has **name/value** couple attributes.

The SimpleMatlab task accepts the following parameters:

- **script** : defines which matlab script will be launched. The value attributes will contain the matlab script code (useful for one line scripts only).
- **scriptFile** : defines which matlab script will be launched. The file at the given path will be loaded.
- **scriptUrl** : defines which matlab script will be launched. The file at the given remote url will be loaded.

- **input** : defines an input script which will be launched before the actual matlab script. The value attribute needs to contain the script code (which must be single-line only).

```

<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:0.91
http://proactive.inria.fr/schemas/jobdescriptor/0.91/schedulerjob.xsd"
  xmlns="urn:proactive:jobdescriptor:0.91" id="Matlab_job_simplest">
  <description>A simple Matlab job, which computes the roots of several polynomials</
5.description>
  <taskFlow>
    <task id="root1" preciousResult="true">
      <description>Calculates the root of a polynomial</description>
      <selection>
        <script>
          <file url="http://proactive.inria.fr/userfiles/file/scripts/checkMatlab.js"
/>
        </script>
      </selection>
      <javaExecutable class="org.ow2.proactive.scheduler.ext.matlab.SimpleMatlab">
        <parameters>
          <parameter name="input" value="in=[1 0 3 --2 5 1];"/>
          <parameter name="script" value="out=roots(in);"/>
        </parameters>
      </javaExecutable>
    </task>
    <task id="root2" preciousResult="true">
      <description>Calculates the root of a polynomial</description>
      <selection>
        <script>
          <file url="http://proactive.inria.fr/userfiles/file/scripts/checkMatlab.js"
/>
        </script>
      </selection>
      <javaExecutable class="org.ow2.proactive.scheduler.ext.matlab.SimpleMatlab">
        <parameters>
          <parameter name="input" value="in=[1 0 3 --2 5 2];"/>
          <parameter name="script" value="out=roots(in);"/>
        </parameters>
      </javaExecutable>
    </task>
    <task id="root3" preciousResult="true">
      <description>Calculates the root of a polynomial</description>
      <selection>
        <script>
          <file url="http://proactive.inria.fr/userfiles/file/scripts/checkMatlab.js"
/>
        </script>
      </selection>
      <javaExecutable class="org.ow2.proactive.scheduler.ext.matlab.SimpleMatlab">
        <parameters>
          <parameter name="input" value="in=[1 0 3 --2 5 3];"/>
          <parameter name="script" value="out=roots(in);"/>
        </parameters>
      </javaExecutable>
    </task>
    <task id="root4" preciousResult="true">
      <description>Calculates the root of a polynomial</description>
      <selection>
        <script>
          <file url="http://proactive.inria.fr/userfiles/file/scripts/checkMatlab.js"
/>
        </script>
      </selection>
      <javaExecutable class="org.ow2.proactive.scheduler.ext.matlab.SimpleMatlab">
        <parameters>
          <parameter name="input" value="in=[1 0 3 --2 5 4];"/>
          <parameter name="script" value="out=roots(in);"/>
        </parameters>
      </javaExecutable>
    </task>
  </taskFlow>
</job>

```

Example 5.1. Simple Matlab Job descriptor Example

5.3. A More Complex Example : a Matlab task flow

Now we will get through a more complex example. This example will use an interesting feature of the Matlab extension : the ability to pass results of one task as inputs of another task. This exemple, on the contrary of the previous one, is not a simple parallel batch processing, it's a flow of tasks, which depends from each others.

This example will compute the sum of a big, randomly-generated array, values taken from -50 to +50. The example contains 3 steps:

1. It splits the big array into several smaller arrays.
2. It computes the sum of each array in parallel.
3. It merges the results from each parallel sub-total to compute the final sum.

This is not, of course, a real-case example as computing the sum of a big array will be much faster on a single machine (due to the overhead of launching Java and a Matlab engine and the network latency), but it is meant to illustrate a simple task flow in Matlab.

We'll go through the new concepts introduced in this example compared to the previous one. Have a look at the new job descriptor first:

```

    <?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:0.91
http://proactive.inria.fr/schemas/jobdescriptor/0.91/schedulerjob.xsd"
  xmlns="urn:proactive:jobdescriptor:0.91" id="Matlab_job" priority="normal" logFile=
"${HOME}/matlab_job.log" >
  <description>A simple Matlab taskflow, which computes the sum of a randomly-generated
array</description>
  <variables>
    <variable name="PROACTIVE_HOME"
      value="/home/user/ProActive" -/>
    <variable name="MATLAB_SCRIPTS"
      value="${PROACTIVE_HOME}/scripts/unix/matlab/examples" -/>
  </variables>
  <taskFlow>
    <task id="splitter">
      <description>Splits a big array</description>
      <selection>
        <script>
          <file url="http://proactive.inria.fr/userfiles/file/scripts/checkMatlab.js"
/>
        </script>
      </selection>
      <javaExecutable class="org.ow2.proactive.scheduler.ext.matlab.MatlabSplitter">
        <parameters>
          <parameter name="scriptFile" value="${MATLAB_SCRIPTS}/splitter.m"/>
          <parameter name="input" value="in=round(rand(1,1000000)*100-50)/>
          <parameter name="number_of_children" value="6"/>
        </parameters>
      </javaExecutable>
    </task>
    <task id="sum1">
      <description>Calculates the sum of the elements in the array</description>
      <depends>
        <task ref="splitter"/>
      </depends>
      <selection>
        <script>
          <file url="http://proactive.inria.fr/userfiles/file/scripts/checkMatlab.js"
/>
        </script>
      </selection>
      <javaExecutable class="org.ow2.proactive.scheduler.ext.matlab.SimpleMatlab">
        <parameters>
          <parameter name="index" value="0"/>
          <parameter name="scriptFile" value="${MATLAB_SCRIPTS}/summer.m"/>
        </parameters>
      </javaExecutable>
    </task>
    <task id="sum2">
      <description>Calculates the sum of the elements in the array</description>
      <depends>
        <task ref="splitter"/>
      </depends>
      <selection>
        <script>
          <file url="http://proactive.inria.fr/userfiles/file/scripts/checkMatlab.js"
/>
        </script>
      </selection>
      <javaExecutable class="org.ow2.proactive.scheduler.ext.matlab.SimpleMatlab">
        <parameters>
          <parameter name="index" value="1"/>

```

Example 5.2: Complex Matlab Job descriptor

5.3.1. Descriptor variables

The **variables** declaration allows a user to define a variable which can be used as a pattern in other parts of the descriptor. This helps writing more generic descriptors and replacing only the variables values to adapt the descriptor to many contexts. Here is the variables declaration in the preceding descriptor:

```
<variables>
  <variable name="HOME" value="/user/fviale/home"/>
  <variable name="MATLAB_SCRIPTS" value="{HOME}/matlab"/>
</variables>
```

the **variables** tag contains a list of **variable** tags which each defines a variable through a **name** and a **value** attribute. The variable can then be used by writing the pattern **{name_of_the_variable}**. Variable can be reused inside the variable declaration itself, but the variables are processed sequentially from top to bottom. Therefore, in this example, In this example the MATLAB_SCRIPTS variable could not be used before the HOME variable.

5.3.2. New Tasks : MatlabSplitter and MatlabCollector

```
<javaExecutable class="org.objectweb.proactive.extensions.scheduler.ext.matlab.MatlabSplitter">
  <parameters>
    <parameter name="scriptFile" value="{MATLAB_SCRIPTS}/splitter.m"/>
    <parameter name="input" value="in=round(rand(1,1000000)*100-50)/>
    <parameter name="number_of_children" value="6"/>
  </parameters>
</javaExecutable>

<javaExecutable class="org.objectweb.proactive.extensions.scheduler.ext.matlab.MatlabCollector">
  <parameters>
    <parameter name="scriptFile" value="{MATLAB_SCRIPTS}/collector.m"/>
  </parameters>
</javaExecutable>
```

Two new tasks appear in this descriptor : the **MatlabSplitter** and the **MatlabCollector**. The Splitter task is used to split an input into a list of several chunks. The Collector task is used to collect and merge the results from several parallel tasks. Each of these tasks come with the same parameter as the SimpleMatlab tasks with an addition: The Splitter expects an additional parameter called **number_of_children**. This parameter tells the Matlab script responsible for splitting in how many parts the input should be divided.

5.3.3. Task dependencies

In order to do complex task flows, it is necessary to add the concept of dependencies between tasks.

```
<task name="sum1">
  <description>Calculates the sum of the elements in the array</description>
  <depends>
    <task ref="splitter"/>
  </depends>
  -...</task>
```

The **depends** tag in this task definition defines a dependency of the task named "sum1" to the task named "splitter". This means that the task sum1 will be launched after the task splitter is complete, and that the outputs of splitter will be fed as inputs to sum1.

You'll notice that in this example, all the sumX SimpleMatlab tasks depend from the Splitter. This means that the output from the Splitter will be fed to each sum task. On the other hand, the Collector depends on every sumX task. It will be launched only after all these tasks have completed, and the results of all these tasks will be the inputs of the Collector, you'll see on

5.3.4. New parameter in SimpleMatlab tasks: index

```
<task name="sum1">
  <description>Calculates the sum of the elements in the array</description>
  <depends>
```

```
<task ref="splitter"/>
</depends>
<selection>
  <script>
    <file url="http://proactive.inria.fr/userfiles/file/scripts/checkMatlab.js"/>
  </script>
</selection>
<javaExecutable class="org.objectweb.proactive.extensions.scheduler.ext.matlab.SimpleMatlab"
>
  <parameters>
    <parameter name="index" value="0"/>
    <parameter name="scriptFile" value="${MATLAB_SCRIPTS}/summer.m"/>
  </parameters>
</javaExecutable>
</task>
```

A new parameter appears in this descriptor for the SimpleMatlab task : the **index** . The parameter is related to the splitting mechanism. It can be defined only inside a SimpleMatlab task and has sense only if the Simple task has a Splitter task as parent. The Splitter sends an output in the form of a list of results to each child task. The same list will be sent to every children. Therefore, each one needs to specify at which index of the list it will look at. For example, a splitter task splits the array [1,2,3,4] into two arrays [1,2] and [3,4], the first child needs to specify index 0 and second index 1 (note that the indexes range from 0 to number_of_children-1). By specifying these indexes, the first child will get as input the array [1,2] and the second child will get [3,4].

5.3.5. Matlab Scripts for this example

5.3.5.1. Script of the Splitter Task

```
n = fix(length(in)/nout);
for i = 1:nout-1
    out(i) = {in(1+n*(i-1):n*i)};
end
out(nout) = {in(1+n*(nout-1):end)};
```

The Splitter script contains two important aspects:

- It contains two inputs, the variable **in** which is fed by the "input" script of the splitter task, and the variable **nout** which contains the value of the **number_of_children** parameter.
- The **out** variable, which is the output of the script must be a cell array of size nout.

5.3.5.2. Script of the Summing Task

```
out = sum(in);
```

5.3.5.3. Script of the Collector Task

```
out = 0
for i = 1:length(in)
    out = out + in{i}
end
```

The important aspect of the The Collector script is that the input parameter **in** is a cell array.

Part IV. ProActive Scheduler's Scilab extension

Table of Contents

Chapter 6. ProActive Scheduler's Scilab Extension 50

- 6.1. Presentation 50
- 6.2. Quick Start with the Scilab Extension 50
 - 6.2.1. Installation 50
 - 6.2.2. The Scilab Job descriptor 50

Chapter 6. ProActive Scheduler's Scilab Extension

6.1. Presentation

Scilab is a scientific software for numerical computations. Developed since 1990 by researchers from INRIA and ENPC, it is now maintained and developed by Scilab Consortium since its creation in May 2003. Scilab includes hundreds of mathematical functions with the possibility to add interactively programs from various languages (C, Fortran...). It has sophisticated data structures (including lists, polynomials, rational functions, linear systems...), an interpreter and a high level programming language. Scilab works on most Unix systems (including GNU/Linux) and Windows (9X/2000/XP).

Similarly to ProActive Scheduler's Matlab extension, the goal of the Scilab Extension is to:

- allow users to easily launch Scilab scripts over an heterogeneous grid
- describe complex task flows in Scilab using human-readable XML descriptors
- Ability to communicate the result of one task as input of another task.
- users won't have to write any other code than Scilab script code
- support the following Scilab types : Double, Complex and String Arrays (the extension currently supports fewer types than the Matlab's one).

A good way to start manipulating and scheduling Scilab scripts is to have a look at the standalone (and simple) Matlab/Scilab GUI described in the ProActive documentation. If you want to directly through more complex Scilab job scheduling, go on with the following tutorial.

6.2. Quick Start with the Scilab Extension

We'll write a simple Scilab job example. This example will assume that you are familiar with the example in Section 5.3, "A More Complex Example : a Matlab task flow" . This example will compute the numerical integration of $\sin(x)$ between 0 and π .

6.2.1. Installation

Before starting to use the Scilab interface, you need to install Scilab in your environment. You'll find all the instructions on PROACTIVE/scripts/unix/scilab/README_Scheduler (Scilab section).

Once Scilab is installed, you won't need at runtime to bother where Scilab is installed, the Scheduler will determine it for you.

6.2.2. The Scilab Job descriptor

This is the descriptor of the scilab job which will be executed inside the scheduler

```

    <?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href=
"../../src/Extra/org.ow2.proactive.scheduler/common/xml/stylesheets/variables.xsl"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:0.91
http://proactive.inria.fr/schemas/jobdescriptor/0.91/schedulerjob.xsd"
  xmlns="urn:proactive:jobdescriptor:0.91" id="Scilab_job"
  priority="normal" logFile="${HOME}/scilab_job.log">
  <description>
    A simple Scilab batch job, which computes the numerical
    integration of sin(x) between 0 and PI
  </description>
  <variables>
    <variable name="PROACTIVE_HOME"
      value="/home/user/ProActive" -/>
    <variable name="SCILAB_SCRIPTS"
      value="${PROACTIVE_HOME}/scripts/unix/scilab/examples" -/>
  </variables>
  <taskFlow>
    <task id="t1">
      <description>
        Calculates the numerical integration of sin(x) between
        i*PI/n and (i+1)*PI/n
      </description>
      <selection>
        <script>
          <file
            url="http://proactive.inria.fr/userfiles/file/scripts/checkScilab.js" -/>
          </script>
        </selection>
        <javaExecutable
          class="org.ow2.proactive.scheduler.ext.scilab.SimpleScilab">
          <parameters>
            <parameter name="scriptFile"
              value="${SCILAB_SCRIPTS}/intsin.sci" -/>
            <parameter name="input" value="i=0;n=5;" -/>
            <parameter name="outputs" value="out" -/>
          </parameters>
        </javaExecutable>
      </task>
      <task id="t2">
        <description>
          Calculates the numerical integration of sin(x) between
          i*PI/n and (i+1)*PI/n
        </description>
        <selection>
          <script>
            <file
              url="http://proactive.inria.fr/userfiles/file/scripts/checkScilab.js" -/>
            </script>
          </selection>
          <javaExecutable
            class="org.ow2.proactive.scheduler.ext.scilab.SimpleScilab">
            <parameters>
              <parameter name="scriptFile"
                value="${SCILAB_SCRIPTS}/intsin.sci" -/>
              <parameter name="input" value="i=0;n=5;" -/>
              <parameter name="outputs" value="out" -/>
            </parameters>
          </javaExecutable>
        </task>
      <task id="t3">
        <description>

```

Example 6.1: Scilab Job descriptor Example

Here is the Scilab script that calculates individual integrals

```
out = integrate('sin(x)', 'x', i*pi/n, (i+1)*pi/n);
```

Example 6.2. Integral script

Here is the Scilab script that merges the individual results and computes the final answer

```
out=out1+out2+out3+out4+out5;
```

Example 6.3. Merging script

This descriptor is very similar to the descriptor Example 5.2, “Complex Matlab Job descriptor Example” . We'll go through the similarities and differences of these two descriptors.

6.2.2.1. Similarities with Matlab job descriptor

- Concepts of job, tasks and dependences are common to all ProActive Scheduler jobs, so we find here the same concepts than in Matlab's.
- Definitions of task's main and input scripts are done through the same task parameters **script** , **scriptFile** , **scriptId** and **input**

6.2.2.2. Differences with Matlab job descriptor

For example in the following task :

```
<javaExecutable
class="org.objectweb.proactive.extensions.scheduler.ext.scilab.SimpleScilab">
  <parameters>
    <parameter name="scriptFile"
      value="{SCILAB_SCRIPTS}/intsin.sci" -/>
    <parameter name="input" value="i=3;n=5;" -/>
    <parameter name="outputs" value="out" -/>
  </parameters>
</javaExecutable>
```

- The main Scilab task is now called **SimpleScilab** . There exists no splitting mechanism yet, but there is a merging mechanism explained below.
- A new important task parameter appears : **outputs** . This parameter is used to specify which variables will be extracted from the Scilab environment at task's end. You can specify multiple output variables by separating them with commas. If you don't specify an output parameter, the variable called **out** will be extracted (leading to an error if it doesn't exist).
- The merging mechanism is different than for Matlab's. If a task depends from a bunch of other tasks. An automatic environment merging will be done. For example if we have 3 tasks A,B,C and C depends of A and B. if task A outputs a variable "a" and task B outputs a variable "b", task C will get as input both variable "a" and "b". Now a problem arise when several tasks output the same variable name. In order to avoid overlapping, and to allow merging of results, this variable will be renamed by appending index at the end of the conflicting variable name. In the current example, each tasks t1 - t5 output the same variable "out". task t6 will accordingly get as input variable out1 - out5. The index starts from 1 and the order matches the depends list order.