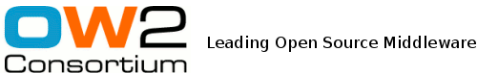
An Open Source Middleware For Parallel, Distributed, Multicore Computing

ProActive Scheduling

Version 2011-03-24

The OASIS Research Team and ActiveEon Company



Generated on 2011-03-24

ProActive Scheduling v2011-03-24 Documentation

Legal Notice

Contact: proactive@ow2.org or contact@activeeon.com

Mailing List

proactive@ow2.org

Mailing List Archive

http://www.objectweb.org/wws/arc/proactive

Bug-Traking System

http://bugs.activeeon.com/browse/PROACTIVE

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
e-mail: Denis.Caromel@inria.fr

Contributors from OASIS Team	Contributors from ActiveEon Company
Brian Amedro	Vladimir Bodnartchouk
Francoise Baude	Arnaud Contes
Francesco Bongiovanni	Cédric Dalmasso
Florin-Alexandru Bratu	Christian Delbé
Viet Dung Doan	Arnaud Gastinel
Yu Feng	Jean-Michel Guillaume
Imen Filali	Olivier Helin
Fabrice Fontenoy	Clément Mathieu
Ludovic Henrio	Maxime Menant
Fabrice Huet	Emil Salageanu
Elaine Isnard	Jean-Luc Scheefer
Vasile Jureschi	Mathieu Schnoor
Muhammad Khan	
Virginie Legrand Contes	
Eric Madelaine	
Elton Mathias	
Paul Naoumenko	
Laurent Pellegrino	
Guilherme Peretti-Pezzi	
Franca Perrina	
Marcela Rivera	
Christian Ruz	
Bastien Sauvan	
Oleg Smirnov	
Marc Valdener	
Fabien Viale	

Former Important Contributors

Laurent Baduel (Group Communications)
Vincent Cave (Legacy Wrapping)
Alexandre di Costanzo (P2P, B&B)
Abhijeet Gaikwad (Option Pricing)
Mario Leyton (Skeleton)
Matthieu Morel (Initial Component Work)
Romain Quilici
Germain Sigety (Scheduling)
Julien Vayssiere (MOP, Active Objects)

Table of Contents

List of Figures

Part I. ProActive Scheduling

Table of Contents

Chapter 1. Overview

1.1. Overview

Executions of parallel tasks on distributed resources (what we call 'nodes'), such as networks of desktops or clusters, require a main system for managing resources and handling task executions: **a batch scheduler**. A batch scheduler provides an abstraction of resources for users. The scheduler enables users to submit jobs, containing one or several tasks, and then to execute these tasks on available resources. It allows several users to share a same pool of resources and also to manage issues related to distributed environment, such as failing resources. The ProActive Scheduler is connected to a Resource Manager providing therefore the resource abstraction.

The Scheduler is accessible either from a Java programming API or a command-line based job submitter. In the following chapters, we will also present 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 administration shell or the GUI. The graphical user and administration interface which can be plugged in the scheduler core application is presented in Chapter 5, *ProActive Scheduler Eclipse Plugin*). The graphical interface has been developed as an Eclipse RCP Plugin.

1.2. Scheduler Installation

The Scheduler archive contains a sources folder, a distribution folder that contains every library used by the ProActive Scheduler, a bin folder that contains every starting script, and a sample directory including lots of job XML descriptors and scripts examples (see Chapter 2, *User guide* to know what a job descriptor is). More folders are available but these are the ones which are needed to start.

- First of all, start a command shell and go into the **bin/[os]** directory into your installed scheduler home path.
- Then launch the **scheduler-start-clean[.bat]** script to create the database and launch the Scheduler. This database is used to store ProActive Scheduler activities and to offer fault tolerance. The scheduler will start and connect to the Resources Manager that either you have already launched or that the scheduler has created if no Resources Manager has been found. Scheduler starting sequence is finished when Scheduler successfully created on rmi://hostname:port/ is displayed.

  At this point the ProActive Scheduler is started with 4 available nodes.
- What you can do now is submitting a job. To do so, just start the **scheduler-client[.bat]** script with proper parameters (in the same directory). Using **scheduler-client[.bat] -submit ../../samples/jobs_descriptors/Job_8_tasks.xml**, this will request for login and password and then, submit this job to the scheduler. The default login and password, is **user pwd**.

  Once executed, you can see that the scheduler is now scheduling this job. You can also see the Scheduler activity by starting the Scheduler Eclipse Plugin : the Graphical User Interface for the Scheduler. To do so, just uncompress the Scheduler_Plugin archive and start the **Scheduler[.exe]** launcher. The first screen presents a non-connected Scheduler interface. Just right click, then connect. You will be requested for a started Scheduler URL, user name and password. If you followed this quick start step by step, just fill URL field with **rmi//localhost:1099/** where 1099 is the default ProActive port for RMIRegistry. Finally, enter **user** for the user name and **pwd** in the password field. For further information on the GUI, please refer to the Scheduler Eclipse plugin documentation.

Warning

Some features of ProActive Scheduling rely on Java Scripting API (JSR 223), which is bugged under MacOSX/Java 1.5.0_16: some JSR 223 specific classes contained in the AppleScriptEngine.jar (loaded from the boot classpath) are compiled under Java 1.6. As a consequence, using default Java 1.5.0_16 for starting any part of ProActive Scheduling (including graphical clients and worker nodes) can lead to the following exception (if scripts capabilities are used):

java.lang.UnsupportedClassVersionError: Bad version number in .class file

To fix this issue, you must remove or rename the AppleScriptEngine.jar from /System/Library/Java/Extensions directory.

1.3. Scheduler Basics

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**.

There is one type of job, described hereafter:

- **TASKSFLOW** - represents a job containing a bag of tasks, which can be executed either in parallel or according to a dependency tree. Tasks inside this job can be either Java (A task written in Java extending a given interface) or Native (Any native process). Each task can be a single process or an MPI application (Execution starts with a given predefined number of resources on which the user can start the application).

A finished job contains all task results provided by the whole process. However, if you want to highlight a result which is more important, you can mark its corresponding task as **precious** in order to retrieve their result easily in the job result. If the job termination has been caused by a failure, then 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 2.1, “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 2 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, a compiled C/C++ application, a shell or batch script. A native task can be specified by a simple command line, or by a 'generation script' that can dynamically generate the command line to be executed (for instance, according to the computing node's operating system wherein the task is executed).

During its execution, a task can crash due to the host (computing node) or code failure. The Scheduler solves this potential problem by offering parameterizable tasks which can be restarted a set number of times (see maxNumberOfExecution in section Section 2.2, “Create and add a task to a job”).

A task may optionally be defined with 4 kinds of scripts (selection-script, pre-script, post-script and cleaning-script) which allow selection and configuration of suitable resource for a given task before and after task execution (see Section 2.2, “Create and add a task to a job”).

Dependencies between tasks can also be defined. This aspect is detailed in the 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 result of an ancestor task to its children as parameter. Dependency between tasks is then both a temporal dependency and a data dependency.

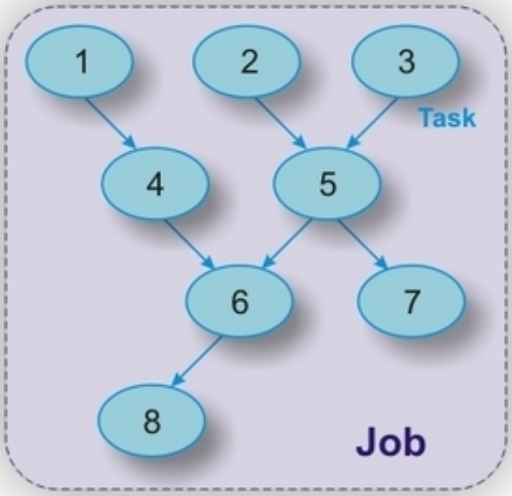


Figure 1.1. Taskflow job example

In this example, an 8 task job is defined (where job's type is TaskFlow). As you can see, task 4 depends on task 1, task 5 depends on tasks 2 and 3, and so on... In other words, task 4 will wait for the end of task 1 before starting and task 5 will wait for the end of task 2 **AND** 3... In addition, the order in which you specify that task 5 depends on 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 tasks will therefore be given to task 5 (as parameter) in this order.

1.3.4. Scheduling Policy

By default, the Scheduler will schedule tasks according to the default **FIFO** (First In First Out) priority policy. If a job needs to be scheduled quickly, increase its priority, or ask your administrator for another policy.

Chapter 2. User guide

2.1. 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 is currently possible to create one type of job. A job can be created using an XML descriptor or the provided ProActive Scheduler Java API or also with a simple flat file.

2.1.1. Job XML descriptor

The Job XML descriptor is created following the schema given in appendix (Chapter 11, *XSD Job Schema*).

Several parameters can be set for a job:

- **name** - name of the job.
- **projectName** (optional) - name of the project. This information provide a way to gather different jobs.
- **priority** (optional - normal by default) - scheduling priority level of the job. A user can only set the priority of his jobs and can only use the values 'lowest', 'low', or 'normal'. There are two higher priority levels 'high' and 'highest' which can be only set by the administrator.
- **cancelJobOnError** (optional - false by default) - defines whether the job must continue when a user exception or error occurs during the job process. This property can also be defined in each task. If the value of this property is defined at the job level, each cancelJobOnError property of each task will have this value as the default one (excepted if this property has been set at the task level). 'True' implies for the job to immediately stop every remaining running tasks if an error occurs in one of the tasks. It is useful when there is no need to go further after a task failure.
- **restartTaskOnError** (optional - anywhere by default) - defines whether tasks that have to be restarted will restart on an other resource. Defining this property will set the restartTaskOnError property of each task to this value as the default one (excepted if this property has been set at the task level). Possible values are 'anywhere' or 'elsewhere' meaning respectively that the concerned task will be restart on any available resources or especially on a different one. A task can be restarted when an exception occurred (Java Task) or an error code is returned (Native Task).
- **maxNumberOfExecution** (optional - 1 by default) - defines how many times tasks are allowed to be restarted. Defining this property will set the nbMaxOfExecution property of each task to this value as the default one (excepted if this property has been set at the task level). The value has to be a non-negative and non-null integer.
- **logFile** (optional) - path of a log file. Set it if you want to save the job generated logs (STDOUT and STDERR) in a file. **As this file is created and filled by the scheduler server, the path must be also reachable from the scheduler server (not only from the client).**
- **variables** (optional) - variables which can be reused throughout the descriptor file. Inside this tag, each variable can be reused (even in another following variable definition) by using the syntax ${name_of_variable}. Note that you can also refer to variables defined in the JVM properties. For instance, if the JVM that starts the Job parser (JobFactory) has been started with the option -Dtoto=helloWorld, the variable ${toto} in the XML descriptor file would have 'helloWorld' as value. This way is not commonly used but can be very useful to manage your relative path.
- **description** (optional) - human readable description of the job.
- **genericInformation** (optional) - defines some information inside your job. These information can be read by the policy of the Scheduler and can be used to modify the scheduling behavior. As an example, the administrator can set the policy to be influenced by this information.
- **jobClasspath** (optional) - equivalent to the Java classpath. All classes in this path can be loaded by Java tasks contained in this job. The jobClasspath can either contain class directories or jar files.
- **inputSpace** (optional) - URL that defines the INPUT space of the job. The INPUT space URL represents an abstract (or real) link to a real dataSpace. It is used for data transfer and offers a way to get files to be computed on the task execution side. (see Section 2.3, “Handling data and files using Data Spaces” for details)

- **outputSpace** (optional) - URL that define the OUTPUT space of the job. The OUTPUT space URL represents an abstract (or real) link to a real dataSpace. It is used for data transfer and offers a way to put produced files from the task execution side to this OUTPUT space. (see Section 2.3, “Handling data and files using Data Spaces” for details)

Warning

The jobClasspath mechanism relies on the file transfer. The size of the jobClasspath files therefore has an impact on performances, especially on the time spent on submitting jobs.

Here is an example of a job descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
   name="job_name" priority="normal" projectName="project_name" cancelJobOnError="true" logFile="path/to/
a/log/file.log">
   <description>Job description</description>
   <jobClasspath>
      <pathElement path="/path/to/my/classes/"/>
      <pathElement path="/path/to/my/jarfile.jar"/>
   </jobClasspath>
   <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>
```

This example does not contain the job type definition. We detail the job type definition in the next sections.

To create a TaskFlow Job, please refer to the next part.

2.1.2. Create a Task Flow job using an XML descriptor

A Task Flow job is a job that can contain one or several task(s) with optional dependencies. To specify that a job is a Task Flow Job, you have to add the 'taskFlow' tag. The previous would therefore be written as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
   name="job_name" priority="normal" projectName="project_name" cancelJobOnError="true" logFile="path/to/
a/log/file.log">
   <description>Job description</description>
   <jobClasspath>
      <pathElement path="/path/to/my/classes/"/>
      <pathElement path="/path/to/my/jarfile.jar"/>
   </jobClasspath>
```

```
  <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>
```

The creation and addition of tasks is described in Section 2.2, “Create and add a task to a job”.

2.1.3. Create a Task Flow job using the Java API

To make a new instance of a TaskFlow job, you have to create a new TaskFlowJob object:

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

2.1.4. Set job parameters using the Java API

Job parameters created through the Java API are identical to those created through XML files (see Section 2.1.1, “Job XML descriptor”). The example below creates a TaskFlow job using the Java Scheduler API and sets some parameters for the job:

```
TaskFlowJob job = new TaskFlowJob();
job.setName("job name");
job.setPriority(JobPriority.NORMAL);
job.setCancelJobOnError(false);
job.setLogFile("path/to/a/log/file.log");
job.setDescription("Job description");
job.addGenericInformation("var1","val1");
job.addGenericInformation("var2","val2");
JobEnvironment je = new JobEnvironment();
je.setJobClasspath(new String[]{"/path/to/my/classes/","/path/to/my/jarfile.jar"});
job.setEnv(je);
```

To create and add tasks to your job, please refer to Section 2.2, “Create and add a task to a job”.

2.1.5. Create a job from a simple flat file

ProActive Scheduler provides a way to define a job with a simple plain text file. This method is simpler than an XML file but presents less features.

If you just need to launch a set of native tasks in parallel, **without dependancies between tasks**, you can create a simple text file and write a list of native commands to submit:

```
# a flat text file containing native commands to launch
# one command per line
```

```
#
# One rule to know:
# command paths must be absolute paths

/path/to/my/exec/myCmd.sh 1
/path/to/my/exec/myCmd.sh 2
/path/to/my/exec/myCmd.sh 3
/path/to/my/exec/myCmd.sh 4
```

This file represents a job made of 4 native tasks, i.e. 4 native commands to launch. Each line beginning with a '#' is a comment, and is not taken into account. You have to put one native command per line, and commands must be absolute paths. Contrary to a job definition in XML, you cannot specify any dependencies between tasks. This job definition is useful for embarrassingly parallel jobs. Many features of job are not available at creation time, but can be specified at submission time: log file, job name, and selection scripts.

To see how to submit this kind of job descriptor, see Section 2.6.2, “Submit a job using the shell command”.

2.2. 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.

2.2.1. Create and add a Java task

Note : A java task can only be added to a TaskFlow Job.

To learn how to create a TaskFlow job, pleas refer to Section 2.1.2, “Create a Task Flow job using an XML descriptor” or Section 2.1.3, “Create a Task Flow job using the Java API”. Once your TaskFlow job created, you can add as many Java tasks as needed to perform an application.

2.2.1.1. Define your own Java executable

You can create your own java executable by implementing scheduler executable interfaces. In this sens, 'executable' means the executed process (that is a Java class in this case). Here is an example on how to create your own Java executable:

```
public class WaitAndPrint extends JavaExecutable {

  @Override
  public Serializable 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.hadException()) {
          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("Task terminated");

        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. The way to give arguments to the task will be explain later. Let's see how you can give a foo, a bar and a test argument to the previous example:

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

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

The previous code is given as an example. The default behavior of the init() method is to associate declared Java variables to their value, assuming that the variable names given in arguments (in the map) are the same as the Java declared ones. Thus, The following code behaves exactly like the previous one :

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

//1
@Override
public void init(Map<String, Serializable> args) {
    super.init(args);
}
//2
//commenting from 1 to 2 is also the same of course
```

To sum up, create an executable is just extending the JavaExecutable abstract class, and filling the execute method. The given TaskResult... results argument enables 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 Serializable 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 personal arguments or do some operations before starting the real execution of the task.

Moreover, the getNodes() method retrieves the list of nodes define in the description of the task. It is a way to make MPI or ProActive application on the given resources. Here's an example of a ProActive application made in a multi-nodes JavaExecutable :

```
public class ProActiveExample extends JavaExecutable {

   private int numberToFind = 5003;

   @Override
   public Serializable execute(TaskResult... results) {
      System.out.println("Multi-node started !!");

      ArrayList<Node> nodes = getNodes();

      // 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;
   }
```

```
  private class Controller {
    // Managed workers
    private Vector<Worker> workers;

    /**
     * Create a new instance of Controller.
     *
     * @param workers
     */
    public Controller(Vector<Worker> workers) {
      this.workers = workers;
    }

    // start computation
    /**
     * Find the Nth prime number.
     *
     * @param nth the prime number to find
     * @return the Nth prime number.
     */
    public int findNthPrimeNumber(int nth) {
      long startTime = System.currentTimeMillis();
      BooleanWrapper flase = new BooleanWrapper(false);
      int found = 0;
      int n = 2;

      while (found < nth) {
        Vector<BooleanWrapper> answers = new Vector<BooleanWrapper>();

        // send requests
        for (Worker worker : workers) {
          BooleanWrapper resp = worker.isPrime(n);
          answers.add(resp);
        }

        PAFuture.waitForAll(answers);

        if (!answers.contains(flase)) {
          workers.get(found % workers.size()).addPrimeNumber(n);
          System.out.println("--->" + n);
          found++;
        }

        n++;
      }

      long stopTime = System.currentTimeMillis();
      System.out.println("Total time (ms) " + (stopTime - startTime));

      return n - 1;
    }
  }

}
```

2.2.1.2. Create and add a Java task using an XML descriptor

The task is the entity that will be scheduled by ProActive Scheduler. As it has been explained in 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 either be created using an XML descriptor or using the provided ProActive Scheduler Java API. This section deals with the creation of a task and its addition to a job.

Take a look at the following example to understand the syntax of a task:

```
<task name="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 name="task2">
  <depends>
    <task ref="task1"/>
  </depends>
  <parallel numberOfNodes="3"/>
  <javaExecutable class="org.ow2.proactive.scheduler.examples.MultiNodeExample">
    <parameters>
      <parameter name="numberToFind" value="100"/>
    </parameters>
  </javaExecutable>
</task>
```

The Java task is composed of a 'javaExecutable' tag which specifies the Java executable class to be used. A set of parameters can also be defined as it has been done for this example. These parameters will be available into the Map of the init(Map) 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 only once 'task 1' has finished. To put these two tasks inside your TaskFlow job, just put it between the 'taskFlow' tags. You may also notice that the 'task 2' will use 3 nodes to be started. So the 'task 2' have to wait for 'task 1' to be terminated and 3 available nodes. Here is a complete ready-to-be-scheduled TaskFlow job:

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
  name="job_name" priority="normal" projectName="project_name" cancelJobOnError="true" logFile="path/to/
a/log/file.log">
  <description>Job description</description>
  <jobClasspath>
    <pathElement path="/path/to/my/classes/"/>
    <pathElement path="/path/to/my/jarfile.jar"/>
  </jobClasspath>
  <variables>
    <variable name="val1" value="toto"/>
  </variables>
```

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

  <taskFlow>

    <task name="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 name="task2">
      <depends>
        <task ref="task1"/>
      </depends>
      <parallel numberOfNodes="3"/>
      <javaExecutable class="org.ow2.proactive.scheduler.examples.MultiNodeExample">
        <parameters>
          <parameter name="numberToFind" value="100"/>
        </parameters>
      </javaExecutable>
    </task>

  </taskFlow>
</job>
```

It is obviously possible to mix Java (with one or more nodes) and Native task inside a taskFlow Job. Some other parameters and options can be set into a Java task. The two following examples expose some of them:

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

```
<task name="PI_Computation" walltime="00:10" >
    <genericInformation>
      <info name="name1" value="val1"/>
    </genericInformation>
    <javaExecutable class="org.ow2.proactive.scheduler.examples.MonteCarlo" >
      <forkEnvironment javaHome="">
        <jvmArgs>
          <jvmArg value="-d12"/>
        </jvmArgs>
      </forkEnvironment>
      <parameters>
        <parameter name="steps" value="20"/>
        <parameter name="iterations" value="100000000"/>
      </parameters>
    </javaExecutable>
</task>
```

To have an exhaustive list of available options and theirs functions, please refer to the task explanation section (see Section 2.2.3, “Tasks options and explanations”).

2.2.1.3. Create and add a Java task using the Java API

To **create a Java task**, you first have to instantiate a **JavaTask** object and, then, to specify the class you want to be executed by this task (To make your own executable, please refer to the proper section Section 2.2.1.1, “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. Take a look at the example hereafter so as to see how to use the Java API to create a Java task (also see the Java Documentation of the Scheduler to learn more):

```
//create a Java Task with the default constructor
JavaTask aTask = new JavaTask();
//add executable class or instance
aTask.setExecutableClassName("org.ow2.proactive.scheduler.examples.WaitAndPrint");
//then, set the desired options
aTask.setName("task 1");
aTask.setDescription("This task will do something...");
aTask.addGenericInformation("key","value");
aTask.setMaxNumberOfExecution(3);
aTask.setRestartTaskOnError(RestartMode.ELSEWHERE);
aTask.setCancelJobOnError(false);
aTask.setResultPreview(UserDefinedResultPreview.class);
//add arguments (optional)
aTask.addArgument("foo",new Boolean(true));
aTask.addArgument("bar",new Integer(12));
```

```
aTask.addArgument("test","test1");

//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 cleaning script
aTask.setCleaningScript(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);
```

Once this done, you still have to add your task to your job:

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

Here are 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 described above)
//then specify dependencies by using the addDependence(Task) method
task5.addDependence(task2);
task5.addDependence(task3);
//or use the addDependences(list<Task>) method as shown
//task5.addDependences(new ArrayList<Task>(task2,task3));

//set a walltime
aTask.setWallTime(10000);
//Define a fork environment to fork the executable (can be empty)
ForkEnvironment env = new ForkEnvironment();
env.setJavaHome("Your/java/home/path");
env.addJVMArgument("-d12");
aTask.setForkEnvironment(env);
```

If a forkEnvironment is set (even empty), the java executable will be started on a brand new JVM with new environment. If a walltime is defined, so the task will automatically be forked. This allows the Scheduler to kill the java process at the end of walltime.

To have an exhaustive list of available options and theirs functions, please refer to the task explanation section (see Section 2.2.3, “Tasks options and explanations”).

2.2.2. Create and add a native task

Note: A native task can only be added to a TaskFlow Job.

To learn how to create a TaskFlow Job, please refer to Section 2.1.2, “Create a Task Flow job using an XML descriptor” or Section 2.1.3, “Create a Task Flow job using the Java API”. Once your TaskFlow Job has been created, you can add as many native tasks as needed to perform your application. A native task can be any native application such as programs, scripts, processes...

2.2.2.1. Create and add a native task using an XML descriptor

Take a look at the following example to understand the syntax of a native task:

```
<!-- This native task example shows a native executable directly started as a command. -->
<task name="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 name="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' tag that specified the executable process to use. A set of parameters has also be defined to provide the executable with some arguments. These arguments will be appended (according to the Runtime.exec() method) 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 only once 'task1_native' has finished. To put these two tasks inside your TaskFlow job, just put that piece of code into 'taskFlow' tags. Here is a complete ready-to-be-scheduled TaskFlow Job:

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
  name="job_name" priority="normal" projectName="project_name" cancelJobOnError="true" logFile="path/to/
a/log/file.log">
  <description>Job description</description>
  <jobClasspath>
    <pathElement path="/path/to/my/classes/"/>
    <pathElement path="/path/to/my/jarfile.jar"/>
  </jobClasspath>
  <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 name="task1_native" maxNumberOfExecution="2">
      <description>Will display 10 dots every 1s</description>
      <nativeExecutable>
        <!-- Consider that the ${WORK_DIR} has been defined in the job description as describe in the job creation
 section -->
        <staticCommand
          value="${WORK_DIR}/native_exec">
          <arguments>
            <argument value="1"/>
          </arguments>
        </staticCommand>
      </nativeExecutable>
    </task>
    <task name="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>
```

To have an exhaustive list of available options and theirs functions, please refer to the task explanation section (see Section 2.2.3, “Tasks options and explanations”).

As it has been said in Section 2.2.1.2, “Create and add a Java task using an XML descriptor”, it is possible to mix Java and Native task inside a taskFlow Job. Please refer to this section for more information.

2.2.2.2. Create and add a native task using the Java API

To **create a native task**, you first have to instantiate a **NativeTask** object and, then, to specify the command corresponding to your native task. This command is actually a String array whose first element is the path to the command and the following ones represent its arguments. Take a look at the following example to see how to create a native task using the Java API (see also Java Documentation of the ProActive Scheduler to learn more):

```
//create a native task with the default constructor
NativeTask aTask = new NativeTask();
//set the command line with its parameters
aTask.setCommandLine(new String[]{"/path/to/command/cmd","param1","param2");
//then, set the desired options:
aTask.setName("task 1");
aTask.setDescription("This task will do
something...");
aTask.addGenericInformation("key","value");
aTask.setPreciousResult(true);
aTask.setMaxNumberOfExecution(3);
aTask.setRestartTaskOnError(RestartMode.ELSEWHERE);
aTask.setResultPreview(UserDefinedResultPreview.class);

//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 cleaning script
aTask.setCleaningScript(script);

//same construction for the  selection script
//the last parameter is used to specified whether the script is static
//or dynamic
SelectionScript selScript = new SelectionScript(script, true);
aTask.setSelectionScript(selScript);
```

Once this done, you still have to add your task to your job:

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

Here are 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 addDependences(list<Task>) method as shown
//task5.addDependences(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 is a last example that describes how to create a native task with a **dynamic command**, that is, a command 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 a task execution. The returned string will be parsed and transform as a String array according to this example ('%' is the escape character):

- the string "/path/to/cmd arg1 arg 2 arg% 3 arg%%% 4 5"
- will generate : [/path/to/cmd,arg1,arg,2,arg 3,arg% 4,5]

where the first element is the command.

```
//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 available options and theirs functions, please refer to the task explanation section (see Section 2.2.3, “Tasks options and explanations”).

2.2.3. Tasks options and explanations

As it has been shown in the different examples, it is possible to create 2 types of tasks. These 2 types have some common features like name, description, scripts and so on. Here are details on each of these common features:

- **name** - name of the task. It can be whatever you want. This name must be unique.
- **description** (optional) - human readable description of the task. It is for human use only. This field is optional but it is better to set it.
- **genericInformation** (optional) - defines some information inside your task. This information could be read inside the policy (similar to job's one). It can be useful to add new complex scheduling behavior.
- **inputFiles** (optional) - selects files to be transfered to the task execution side. It is possible to add a set of files to be copied from the INPUT or OUTPUT space to the worker side. Files can be selected one by one or through regular expressions. For each

selected files, an access mode specify if the files has to be transfered from INPUT, OUTPUT space or if transfer is not needed. (see Section 2.3, “Handling data and files using Data Spaces” for more details).

- **outputFiles** (optional) - selects files to be transfered to the OUTPUT space. It is possible to add a set of files to be copied from the worker side to the OUTPUT space of the job. Files can be selected one by one or through regular expressions. For each selected files, an access mode specify if the files has to be transfered to OUTPUT space or if no transfer is needed. (see Section 2.3, “Handling data and files using Data Spaces” for more details).
- **preciousResult** (optional - false by default) - defines whether a result of a task is important. 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 option to 'true', you will be able to retrieve them easily.
- **cancelJobOnError** (optional - false by default or set to the cancelJobOnError value of its job) - defines whether your job must continue if a user exception or an error occurs during this task process. True means that the job will immediately stop every remaining running tasks if an error occurs in this task. It is useful when there is no need to go further if this task fails. If true, the job will be canceled if the maximum number of execution for this task has been reached.
- **maxNumberOfExecution** (optional - 1 by default or set to the maxNumberOfExecution value of its job) - defines how many times a task will run if it ends with an exception or an error code. If the task ends in a normal way, this property has no effect. If the task has an exception or an error more than defined in this property, there are 2 possibilities:
  - cancelJobOnError=true: then, the job is canceled.
  - cancelJobOnError=false: then, the task ends and the job continues in a normal way.

  In Both cases, the result is the representation of the error (Exception in Java, error code in native).
- **restartTaskOnError** (optional - anywhere by default or set to the restartTaskOnError value of its job) - defines whether this task has to restart on an other resource. Possible values are 'anywhere' or 'elsewhere' meaning respectively that the concerned task will be restart on any available resources or on a different one. A task can be restarted when an exception occurred (Java Task) or when an error code is returned (Native Task) and if the maximum number of execution has not been reached yet. Each time an error occurred, the task is placed in a Faulty(n/m) status and the numberOfExecutionLeft count is decreased. 'n' is the current number of executions and 'm' is the maximum number of executions.

Here is a table that sum-up the different possible executions, that can be useful to know the behavior of your job :

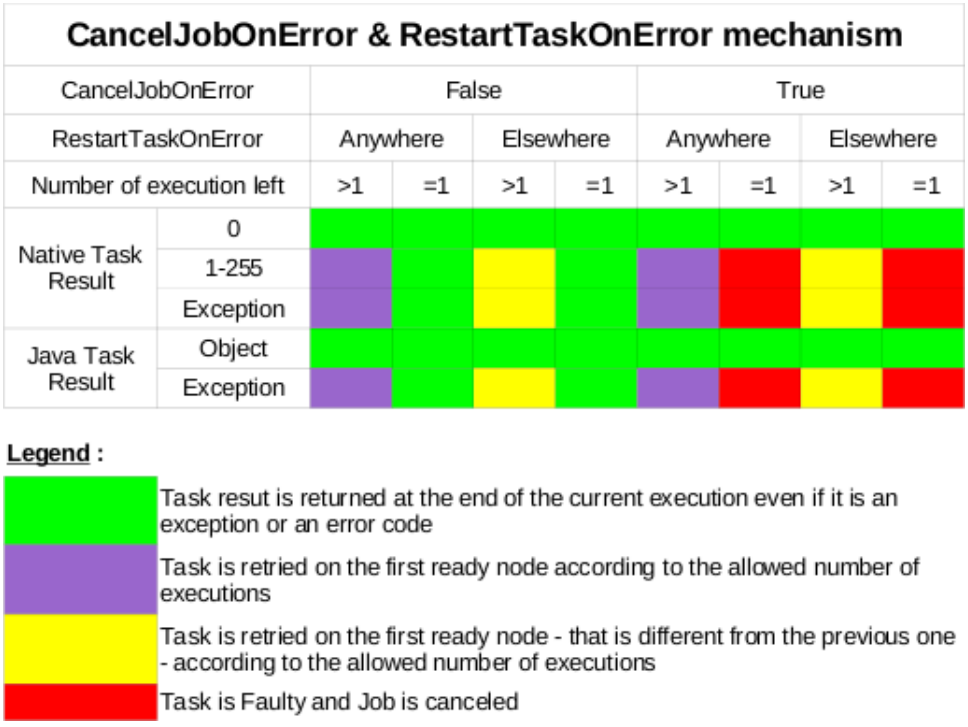


Figure 2.1. CancelJobOnError and RestartTaskOnError behavior

- **parallel** - describes the environment of multi-node task. **numberOfNodes** in this element defines the number of nodes needed by the task (cannot be less than 2).
  - For a java task, one node among the specified number is used to start the task. The other are available through the getNodes() method in the java executable. So, if you need 4 nodes to make your application worked, just ask for 5 nodes. (numberOfNodes="5")
  - For Native task, every asked nodes are available through environment variables :
    - 'PAS_NODESNUMBER' : contains the number of nodes available for the task.
    - 'PAS_NODESFILE' : is the path to a file that contains every available nodes URL (one URL by line)

    Use this environment variable in your own native process to get these informations back.

  parallel allows to define the **topology** of nodes for multi-nodes tasks such as "best proximity", "threshold proximity", "single host exclusive" and others (see the section Section 2.5, “Defining a Topology for Multi-Nodes Tasks” for details).
- **Walltime** (optional) - maximum time for the task execution (timeout). 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. When this property is defined and execution exceeds this time, the task ends with an exception.
- **runAsMe** (optional - default false) - defines if the executable process should be executed under user specific name using its authentication credentials (such as login, password, or ssh key). If the targeted executable is a java executable, setting this argument (true) implies having a forked JVM : It means a new JVM will be created in a process owned by the user himself. For further information about this feature, please go to Section 2.2.4, “Starting a task under a specific user (runAsMe=true)”

  Important Note : With a runAsMe task, it is not possible to propagate properties using the helper class PropertyUtils as explained in the pre/post scripts. Moreover, the nodes file (i.e. the temporary file that contains allocated resources) and the number of allocated nodes are not accessible through PAS_NODESFILE and PAS_NODESNUMBER system environment variables ; those information have to be explictely passed as parameter in the command line respectively as $NODESFILE and $NODESNUMBER.
- **forkEnvironment** (optional, only for Java Executable) - defines whether a task has to be forked and defines its the forked environment. The purpose of a 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) add classpath or redefined Java home path, JVM arguments, sytem environment, working directory or set a script to programmatically configure this fork environment.

  A Forked Java Task is just defined as a Java Task with a forkEnvironment element. For any undefined elements a default empty 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. See Section 2.2.1, “Create and add a Java task” for an illustration of their usage.

  A fork environment can contains the following items :
  - **javaHome** : string containing the Java installation directory that will be used on worker side to start the JVM. (Default is java property "java.home")
  - **workingDir** : string representing the working directory of the newly created JVM. Every relative path will be relative to this directory in the application. Note : the application itself is not started inside this directory.
  - **SystemEnvironment** : list of system "variables" representing the system environment that will be applied to the forked JVM process. A variable can append a previous existing variable, can be created or overwritten.
  - **jvmArgs** : list of "jvmArg" representing every java argument that will be passed to JVM at startup
  - **additionalClasspath** : list of "pathElement" representing the classpath to be added when starting the new JVM.
  - **envScript** : Environment script (javascript, ruby, python) that is able to add/change each items of the fork environment programmatically. This script will be executed on worker side. We provide a variable named "forkEnvironment" which is exactly representing the java object that was set at client side plus the local worker environment. For exemple, if user

set the PATH variable to append "my/path", the fork environment in the script will contain a variable PATH that will be "LOCAL_PATH_VARIABLE:my/path" where LOCAL_PATH_VARIABLE is the PATH content of the process to be started. In this script, the "forkEnvironment" variable can be used exactly as it is in java API.

- **parameters** (optional, only for Java and ProActive Task) - defines some parameters to be transfered to the executable. This is best explained in Section 2.2.1.1, “Define your own Java executable”. Each parameter is define with a name and a value and will be passed to the Java Executable as an Map.
- **arguments** (optional, only for native Task) - defines arguments for your native process. Each argument is define by a value that will be appended to the process name to create a String array 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 (extending the 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, similarily 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.
- **runAsMe** (optional, default is false) - specifify if your task must be executed under your user account or not. If this property is set to 'true', the node on which it will be executed must be configured to accept the fact you can executre a task in your own process. Ask your administrator for more details about the behavior of the nodes. This property also implies you have a valid system account on the execution targeted node machine and you have made your connection to the scheduler using your system password and/or your ssh key if required by the node configuration. See Section 2.6.3, “Submit a job using the Java API” for more details about job submission.
- **scripts** (optional) - The ProActive scheduler supports portable script execution through the JSR 223 Java Scripting capabilities. Scripts can be written in any language supported by the underlying Java Runtime Environment. They are used in the ProActive Scheduler to:
  - Execute some simple Pre, Post and Cleaning processing: optional pre-script, post-script, and cleaning-script
  - Select among available resources the node that suits 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:
  - **selection script** - a 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. A java helper (org.ow2.proactive.scripting.helper.selection.SelectionUtils) is provided for allowing user to simply make some kind of selections. (script samples are available in 'samples/scripts/selection' directory.)
  - **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. Java helper (org.ow2.proactive.scripting.helper.filetransfer package) is also available for file transfer purposes to manage file copies. Some script samples are available in 'samples/scripts/filetransfer' directory.

    Another helper is provided for transmitting Java properties to either forked environment (native or forked Java task) or to all dependent tasks, namely the org.ow2.proactive.scripting.helper.PropertyUtils class. The exportProperty(String name) method makes the property available in the native environment as a system variable (renamed like my.java.property becomes MY_JAVA_PROPERTY). The propagateProperty(String name) makes the property available in the java environment as a Java property (not renamed) of all the dependent tasks. Note that if several value for a single property a propagated to a single task (i.e. a task with several parents), the resulting value is not specified and can be any of the propagated value.
  - **post-script** - the post-script is executed on the same node **after** the task itself in any case. A boolean variable named "success" is available in this script in order to inform user about the success of the task execution. if success is true, the execution has finished successfully, if not, execution has generated an exception. The same helpers than for pre-script can be used in the post-script,

except the org.ow2.proactive.scripting.helper.PropertyUtils.exportProperty(String name) which has no effect since no process can be forked after the post-script.

- **cleaning-script** - the cleaning-script is always executed by the Resource Manager **after** the execution of the task itself or after the post-script (if used). The same helper than for pre-script can be used in the cleaning-script.

For pre, post and cleaning script, you can use the file transfer helper classes. It provides easier way to copy, paste, transfer files from a host to another. (Examples are available in 'samples/scripts/filetransfer' directory)

To use your favorite script language, you just have to add the two needed jars (engine and implementation) in the nodes, Resources Manager and Scheduler classpath.

Note: For any script engine or script implementation used in this package or brought by you, you have to notice that:

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Once your job created, next step is to submit it to the ProActive Scheduler.

2.2.4. Starting a task under a specific user (runAsMe=true)

It is possible to start a task under the job owner if the system is configured for that purpose. Their are 3 possible ways to run the task under user account (in any case, the administrator should have set the workers to authorize one of the 3 methods) :

- **using your Scheduling login** : if workers are configured and user is authorized to run a process under his login without password.
- **using your Scheduling login and password** : if workers are configured and user is authorized to run a process under his login and password.
- **using an ssh key** provided by the administrator : if workers are configured, the administrator should have gave user a sshkey. User must first create a credential containing this key :

  bin/unix/create-cred -F config/authentication/keys/pub.key -l username -p userpwd -k path/to/private/sshkey -o myCredentials.cred : This command will create a new credentials with "username" as login, "userpwd" as password, using scheduler public key at "config/authentication/keys/pub.key" for credentials encryption and using the private ssh key at "path/to/private/sshkey" provided by administrator. The new credential will be stored in "myCredentials.cred"

  Once created, user must connect the scheduler using this credential. (In this case, workers must be under UNIX system)

Note : In the current version and for technical issues, system properties in ForkEnvironment won't be taken into account when "runAsMe" property is true.

2.3. Handling data and files using Data Spaces

The Scheduler infrastructure allows user to handle files during the scheduling process. Based on the ProActive DataSpace, It is now possible to describe from where to pick files, and where to put eventual produced files. This ability supports protocol such as FTP, HTTP, File system, and also distributed file System using the ProActive Provider Server. Here is a brief explanation around the file handling mechanism:

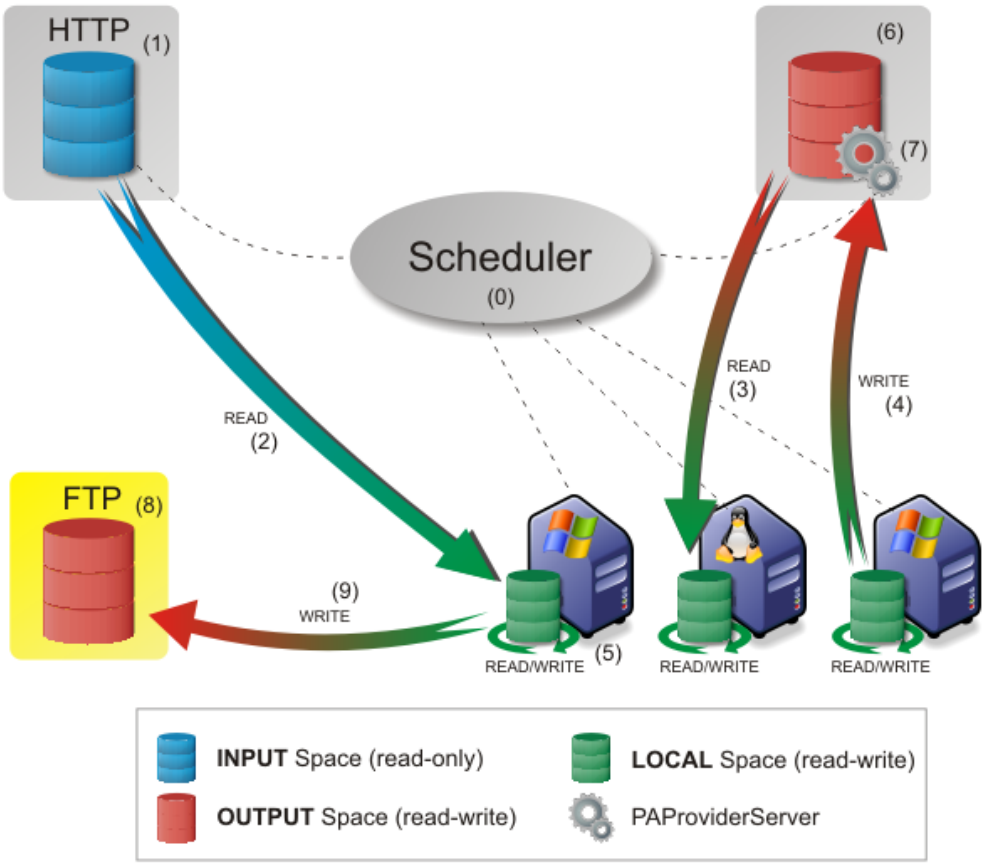


Figure 2.2. Handling data during scheduling

The Scheduler (0) is always connected to default INPUT(1) and OUTPUT(6) spaces, defined by the administrator or in you own job. INPUT space (read-only) is used to store every files and data needed by a job. Each task should be able to read and/or write data in the OUTPUT space (read-write). As there are default INPUT/OUTPUT spaces, users can also define theirs own data spaces by specifying them in the job descriptor. User can define INPUT and/or OUTPUT spaces. If a space is not defined, the default one is used.

As an example and as shown in the image above, an Executable (5) can get files (2) from the INPUT space (1). Once files are located in the LOCAL space, Executable can read and write from/to the LOCAL space. At the end of the task, files will be sent (9) to the OUTPUT (8) FTP space. An other task could read (3) the default OUTPUT space (6), while another could write (4) to the same space (6). Users has to take care about concurrent access in the OUTPUT space.

If using the default spaces, user's data has to be stored into a directory named as his login. This directory will be the root of the space. Ask your administrator if you cannot locate the default targeted spaces.

OUTPUT and INPUT are configured by the user that submits the job, or when not specified by the user, by the Scheduler administrator. Each job execution has an unique INPUT and OUTPUT definition. In addition to these, the GLOBAL space is an input/output space that is always configured by the Scheduler administrator, and is accessible by all nodes. The GLOBAL space can be used as a intermediary storage available for all nodes that is cheaper to acces than the OUTPUT space.

GLOBAL space has to be set by the Scheduler administrator through the **pa.scheduler.dataspace.globalurl** property. The value to use needs to be an URL to a writable space: ie file:/example.com/ or ftp:/example.com/ (HTTP cannot be used as it is read only). Each GLOBAL space directory is unique for a job execution in the scheduler: the virtual path of the GLOBAL space that can be used by a task includes the job id. This means that GLOBAL spaces cannot be used as an initial input space: at the beginning of a job it will always be empty. Additionally, at the end of the job, the GLOBAL subdir for each job will be deleted.

Let's continue with an example of use through a job containing a native task and a Java task.

2.3.1. Using data spaces in a job

A job can define INPUT and/or OUTPUT spaces. To set up a space in the XML descriptor, just proceed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/proactive/jobdescriptor/dev/schedulerjob.xsd"
  name="job_name" priority="normal" projectName="project_name" cancelJobOnError="true" logFile="path/to/a/log/file.log">

  <description>Job description</description>
  ...
  <inputSpace url="http://myserver/directory" />
  <outputSpace url="paprmi://host:1099/myOutputSpace?proactive_vfs_provider_path=/" />
  ...
</job>
```

This example shows how to define both INPUT and OUTPUT spaces. The INPUT space is defined on a HTTP server whereas the OUTPUT one is defined on a remote file system provided by the ProActive data server. The only constraint to take care about is to be sure that the OUTPUT space is a writable space. (for example, it cannot be HTTP protocol).

In the above example, a ProActive data server should be started. To do so, just go into the bin directory and launch the **pa-dataserver[.bat]** script that takes a mandatory argument and a optional one:

- **path**: an absolute path pointing on the local file system directory that will be the root of the started space. (mandatory)
- **name**: the name of the server to be started. Can be useful if more than one server is started on the same host. ('myOutputSpace' in the example).

The same description is abviously possible in job java API:

```
//...
//job is created above ...
job.setInputSpace("http://myserver/directory");
job.setOutputSpace("paprmi://host:1099/myOutputSpace?proactive_vfs_provider_path=/");
//...
```

From now on, every files that will be picked or put in those spaces has to be represented by a "virtual" path whose root is "http://myserver/directory" for the INPUT space, and the path passed (to pa-dataserver) as first argument for the OUTPUT space.

Let's continue with the description of the tasks.

2.3.2. Specifying data spaces in the tasks

A task can define a set of input and output files. Input files can be picked from the job INPUT, OUTPUT or GLOBAL space. To specify a file to be transfered, it is necessary to set two mandatory fields. A third is also useful but is not mandatory:

- **includes**: is a file or a regular expression targeting the abstract relative path to the file you want to select in the job space. For example, "bin/toto*.xml" will target every files starting with 'toto' and ending with '.xml' in the 'bin' directory relative to the chosen space. (mandatory) Note: for read-only and non listable INPUT space (as HTTP), includes files must not be regular expression.
- **accessMode**: specify if the selected files has to be transfered from INPUT or OUTPUT or GLOBAL, to OUTPUT or GLOBAL, or not transfered. (mandatory) Possible values for the input files are: 'transferFromInputSpace', 'transferFromOutputSpace', 'transferFromGlobalSpace', 'none'. Possible values for the output files are: 'transferToOutputSpace','transferToGlobalSpace', 'none'.

- **excludes**: describes a file or every files that must not be targeted among the selected includes files. In our example, if "excludes" is "bin/toto123.xml", the file 'toto123.xml' in the 'bin' directory will finally not be selected. (optional) Note: for read-only (as HTTP) INPUT space, excludes files must not be regular expression.

To set up input and/or output files in the XML descriptor, just proceed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
  name="job_name" priority="normal" projectName="project_name" cancelJobOnError="true" logFile="path/to/
a/log/file.log">

  ...
  <inputSpace url="http://myserver/directory" />
  <outputSpace url="paprmi://host:1099/myOutputSpace?proactive_vfs_provider_path=/" />
  ...

  <taskFlow>
    <task name="task1" retries="2">
      ...
      <inputFiles>
        <files includes="tata" accessMode="transferFromInputSpace"/>
        <files includes="txt/toto*.txt" excludes="txt/toto*2.txt" accessMode="transferFromOutputSpace"/>
        <files includes="dat/tutu*.dat" excludes="dat/tutu*2.dat" accessMode="transferFromGlobalSpace"/>
      </inputFiles>
      <!-- Java or native Executable -->
      <outputFiles>
        <files includes="titi*.dat" excludes="titi*1.dat" accessMode="transferToOutputSpace"/>
        <files includes="titi*.bak" excludes="titi*1.bak" accessMode="transferToGlobalSpace"/>
        <files includes="titi*.txt" accessMode="none"/>
      </outputFiles>
      ...
    </task>
    ...
  </taskFlow>
</job>
```

Until now, LOCALSPACE will be used as the root of the space local to the task execution (green data space on the picture above). In this example, the file 'tata' located at the root of the INPUT space will be transfered from INPUT space to LOCALSPACE and every 'toto*.txt' files excepted 'toto*2.txt' in the 'txt' directory will be transfered from OUTPUT space to LOCALSPACE. Additionally, files matching 'tutu*.txt' excepted 'tutu*2.txt' in the 'dat' directory will be transfered from GLOBAL space to LOCAL space.

At the end of the execution, every produced files 'titi*.dat' excepted the 'titi*1.dat' files will be transfered from LOCALSPACE to OUTPUT space. Additionally, the 'titi*.bak' excepted 'titi*1.bak' will be transferred from LOCAL space to GLOBAL space. The 'titi*.txt' files won't be transfered, it is just identified as an output file by the Scheduler.

Here's the same description using the java API:

```
//task is created above...
//add input files
task.addInputFiles("tata",InputAccessMode.TransferFromInputSpace);
FileSelector fs = new FileSelector(new String[]{"txt/toto*.txt"}, new String[]{"txt/toto*2.txt"});
task.addInputFiles(fs,InputAccessMode.transferFromOutputSpace);
```

```
FileSelector fs2 = new FileSelector(new String[]{"dat/tutu*.txt"}, new String[]{"dat/tutu*2.txt"});
task.addInputFiles(fs2,InputAccessMode.transferFromGlobalSpace);
//...
//add output files
fs = new FileSelector(new String[]{"titi*.dat"}, new String[]{"titi*1.dat"});
task.addOutputFiles(fs,OutputAccessMode.TransferToOutputSpace);
fss = new FileSelector(new String[]{"titi*.bak"}, new String[]{"titi*1.bak"});
task.addOutputFiles(fs,OutputAccessMode.TransferToGlobalSpace);
task.addOutputFiles("titi*.txt",OutputAccessMode.none);
//...
```

Now, let's explain how to get files in the executable.

2.3.2.1. Retrieving files in a native task

To retrieve files as argument in a native task, you have to use the special variable $LOCALSPACE. $LOCALSPACE will be replaced by the real path to the LOCAL space ensuring that your selected input files will be there. To produce files and copy them to the OUTPUT space, just put them at a relative to LOCALSPACE path and select them in the output files list.

The example below shows how to make such a behavior:

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
   name="job_name" priority="normal" projectName="project_name" cancelJobOnError="true" logFile="path/to/
a/log/file.log">

   ...
   <inputSpace url="http://myserver/directory" />
   <outputSpace url="paprmi://host:1099/myOutputSpace?proactive_vfs_provider_path=/" />
   ...

   <taskFlow>
      <task name="task1" retries="2">
         ...
         <inputFiles>
            <files includes="tata" accessMode="transferFromInputSpace"/>
            <files includes="txt/toto*.txt" excludes="txt/toto*2.txt" accessMode="transferFromOutputSpace"/>
         </inputFiles>
         <nativeExecutable>
            <staticCommand value="myExec">
               <arguments>
                  <!-- file 'tata' -->
                  <argument value="$LOCALSPACE/tata"/>
                  <!-- dir 'txt' -->
                  <argument value="$LOCALSPACE/txt"/>
                  <!-- root of LOCALSPACE to store produced files -->
                  <!-- myExec will produced 'titi*' dat and txt files at the root of LOCALSPACE -->
                  <argument value="$LOCALSPACE"/>
               </arguments>
            </staticCommand>
         </nativeExecutable>
         <outputFiles>
```

```
            <files includes="titi*.dat" excludes="titi*1.dat" accessMode="transferToOutputSpace"/>
            <files includes="titi*.txt" accessMode="none"/>
          </outputFiles>
          ...
        </task>
        ...
    </taskFlow>
</job>
```

In this example, the 'myExec' native executable takes one file and one directory as arguments. Every selected input files will be copied into the LOCALSPACE. Once the execution terminated, every produced file in the third argument (LOCALSPACE root) will be copied to the OUTPUT space except the 'titi*.txt' files.

To do the same in java, just proceed as follows:

```
NativeTask t = new NativeTask();
//add input files
t.addInputFiles("tata",InputAccessMode.TransferFromInputSpace);
FileSelector fs = new FileSelector(new String[]{"txt/toto*.txt"}, new String[]{"txt/toto*2.txt"});
t.addInputFiles(fs,InputAccessMode.transferFromOutputSpace);
//add output files
fs = new FileSelector(new String[]{"titi*.dat"}, new String[]{"titi*1.dat"});
t.addOutputFiles(fs,OutputAccessMode.TransferToOutputSpace);
t.addOutputFiles("titi*.txt",OutputAccessMode.none);
t.setName("native_java");
t.setCommandLine(new String[]{"myExec", "$LOCALSPACE/tata", "$LOCALSPACE/txt", "$LOCALSPACE"});
//...
//job.addTask(t);
```

IMPORTANT : At the end of the task, LOCALSPACE is cleared, every files deleted.

2.3.2.2. Retrieving spaces and files in a java task

For java task, everything can be done using the dataspace API inside the javaExecutable. Transferring files works as well, but it is also possible to handle non-transfered files. 6 methods are available in the JavaExecutable to retrieve Spaces or files:

- **getInputSpace()**: retrieves the root of INPUT space. You will get a DataSpacesFileObject on which it is possible to perform some file operations. (Warning ! Files are probably not on the node host)
- **getOutputSpace()**: retrieves the root of OUTPUT space. You will get a DataSpacesFileObject on which it is possible to perform some file operations. (Warning ! Files are probably not on the node host)
- **getGlobalSpace()**: retrieves the root of GLOBAL space. You will get a DataSpacesFileObject on which it is possible to perform some file operations. (Warning ! Files are probably not on the node host)
- **getLocalSpace()**: retrieves the root of the LOCAL space. You will get a DataSpacesFileObject on which it is possible to perform some file operations.
- **getInputFile(pathName)**: selects the file located at "pathName" in the INPUT space. "pathName" has to be relative to the root of the INPUT. You will get a DataSpacesFileObject on which it is possible to perform some file operations. Files can be moved, copied or read. (Warning ! Files are probably not on the node host)
- **getOutputFile(pathName)**: selects the file located at "pathName" in the OUTPUT space. "pathName" has to be relative to the root of the OUTPUT. You will get a DataSpacesFileObject on which it is possible to perform some file operations. File can be moved, copied, read or written. (Warning ! Files are probably not on the node host)
- **getGlobalFile(pathName)**: selects the file located at "pathName" in the GLOBAL space. "pathName" has to be relative to the root of the GLOBAL. You will get a DataSpacesFileObject on which it is possible to perform some file operations. File can be moved, copied, read or written. (Warning ! Files are probably not on the node host)

- **getLocalFile(pathName)**: selects the file located at "pathName" in the LOCAL space. "pathName" has to be relative to the root of the LOCALSPACE. You will get a DataSpacesFileObject on which it is possible to perform some file operations. File can be moved, copied, read or written. (local copy)

Here's a complete JavaExecutable example using the data spaces:

```
public class DSTest extends JavaExecutable {

  @Override
  public Serializable execute(TaskResult... args) throws Throwable {
    //create titi1.dat from tata in the LOCALSPACE as it is copied
    getLocalFile("titi1.dat").copyFrom(getLocalFile("tata"),FileSelector.SELECT_SELF);
    //create titi2.dat -> will be transfered to OUTPUT space at the end
    getLocalFile("titi2.dat").createFile();
    //list toto*.txt in txt dir from LOCALSPACE as they are copied
    //toto*2.txt is not printed as it is exclude
    for (DataSpacesFileObject dsfo : getLocalFile("txt").getChildren()){
      System.out.println(dsfo.getRealURI());
      System.out.println(dsfo.getVirtualURI());
    }
    //create titi3.dat -> will be transfered to OUTPUT space at the end
    getLocalFile("titi3.dat").createFile();
    //create titi12.txt -> won't be transfered to OUTPUT space
    getLocalFile("titi12.txt").createFile();
    //create titi123.txt directly in the OUTPUT space as it won't be transfered
    getOutputFile("titi123.txt").copyFrom(getInputFile("txt/toto12.txt"), FileSelector.SELECT_SELF);
    //as expected, OUTPUT space will contain :
    //titi2.dat : copied automatically
    //titi3.dat : copied automatically
    //titi123.txt : copied manually by the code above
    return "helloWorld";
  }

}
```

As we can see, java executable can read, copy, move files from/to every spaces. At the end of the task, LOCALSPACE is cleared, every files deleted.

2.3.2.3. Retrieving files in a script

Similarly to what is achieved in Section 2.3.2.2, “Retrieving spaces and files in a java task” in Java tasks, Dataspace objects are available in the scripted environments of pre, post and flow scripts (flow scripts are similar to pre/post scripts introduced in Section 2.4, “Enabling Workflows in a Task Flow job”). This allows taking decisions in scripts based, for instance, on the input available to the task or the output that was produced.

A DataSpacesFileObject is bound to the script environment for inputspace, outputspace, globalspace and localspace. The exported variables are respectively **input**, **output**, **globalspace** and **localspace**. Here is an usage example:

```
importPackage(org.objectweb.proactive.extensions.dataspaces.api);
importPackage(java.io);
// output here is a DataSpacesFileObject representing this job's OutputSpace
var f = output.resolveFile("out.dat");
var br = new BufferedReader(new InputStreamReader(f.getContent().getInputStream()));
```

```
var line;
while ((line = br.readLine()) != null) {
  if (line.match(regex) != null) {
    // ... do stuff
  }
}
```

2.4. Enabling Workflows in a Task Flow job

Workflows in Scheduling is a set of Control Flow operations and syntactical constructs that allow any valid Task Flow job to gain expressiveness and dynamic properties.

2.4.1. Use-cases

Figure 2.3, “Simple Workflow use-case” demonstrates the simplest use-case in which Workflows are needed in Scheduling. Without workflows, 6 (split, join, plus one per split image) tasks have to be written statically in the descriptor to achieve this behaviour.

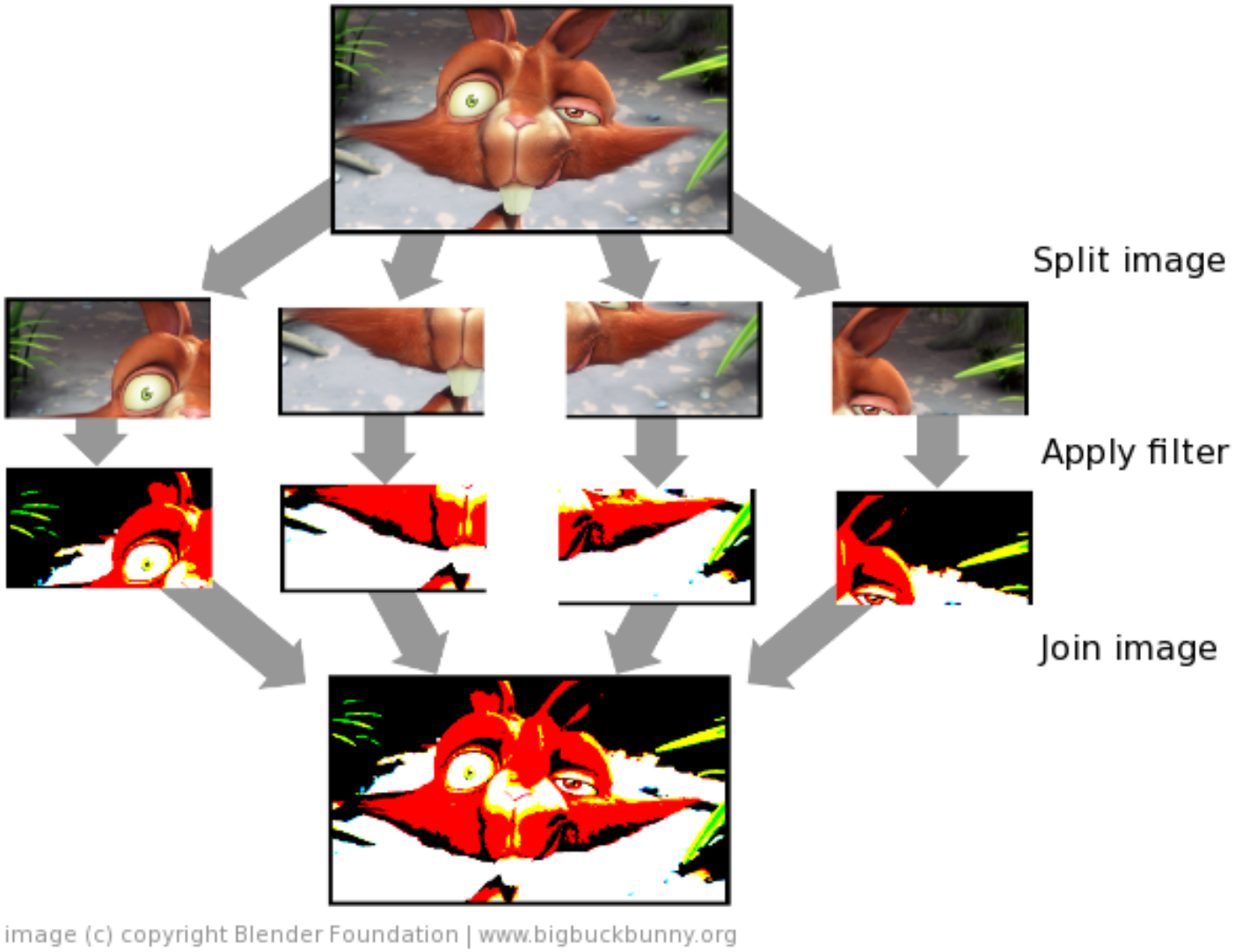


Figure 2.3. Simple Workflow use-case

With Workflows, this job should be composed of only three tasks, and the number of actual split task should be dynamic, and dependant on the size of the input, here an image. The goal of Workflows is, in the end, to provide tools to program logic inside a job for parts that are obviously factorisable.

2.4.2. Specification

Workflows provide three main features: the ability to replicate tasks, the ability to loop on a set of tasks, and the ability to choose between two paths in the flow. These features are defined and restricted by syntactic and semantic rules, in order to ensure the correctness of the job prior to submission.

2.4.2.1. Common rules

- A **dependency** between tasks refers exclusively to an explicit <depends> tag in the descriptor. Other types of links introduced by workflows will use proper vocabulary.
- If a task **t** is started, every task having **t** as dependency should eventually be started.
- A control flow operation is always activated on the compute node, just after the actual task is executed.
- The control flow operation's activation is controlled by a dynamic script using the task's result as input.
- The task on which the action is embedded is called the **initiator**, if a task is used as parameter for an action, it is called a **target**.
- A **task block** is a set of tasks delimited by two tasks tagged **start block** and **end block**, forming a semantic scope.
- **Task blocks** can be nested: each **end block** task matches the last defined **start block** task up in the dependency chain.
- A **Task blocks** can consist of one single task, provided it is neither tagged as **start block** of **end block**.
- Walking up the dependency chain of an **end block** task must always lead to the same **start block** task. Walking down the dependency chain of a **start block** task must always lead to the same **end block** task.
- Three distinct operations are defined: **replicate** (parallelization), **if** (branching) and **loop** (looping).

2.4.2.2. Parallelization

The <replicate> action allows the execution of multiple tasks in parallel when only one task is written in the descriptor, and the number of parallel runs is not statically known.

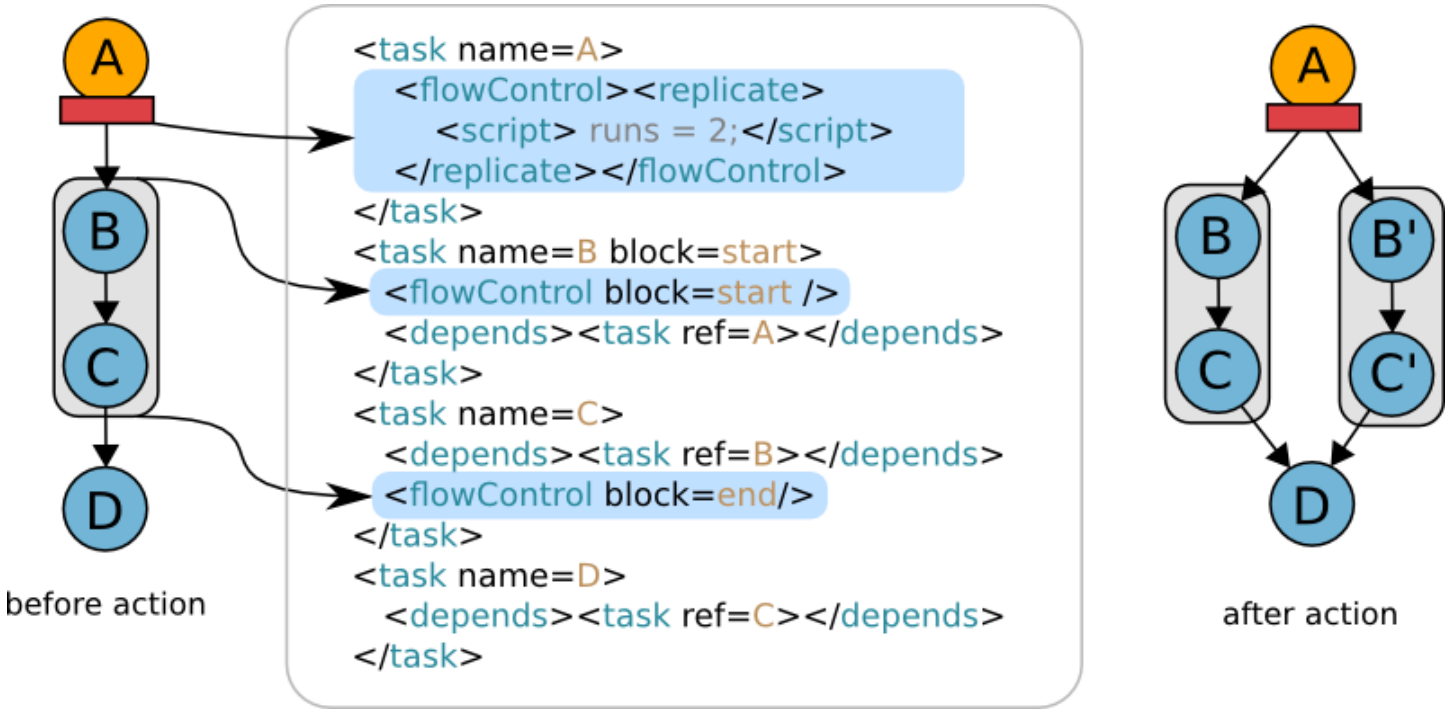


Figure 2.4. Replicate control flow action

- The target is the direct child of the task initiator.
- The initiator can have multiple children; each child is replicated.
- If the target is a **start block**, the whole block is replicated.
- The target must have the initiator as only dependency: the action is performed when the initiator task terminates. If the target has an other pending task as dependency, the behaviour cannot be specified.
- There should always be a merge task after the target of a replicate: if the target is not a start block, it should have at least one child, if the target is a start block, the corresponding end block should have at least one child.
- The last task of a replicated task block (or the replicated task if there is no block) cannot perform a **branching** or **replicate** action.
- The target of a **replicate** action can not be tagged as **end block**.

2.4.2.3. Branching

The **If** action provides the ability to choose between two alternative task flows, with the possibility to merge back to a common flow.

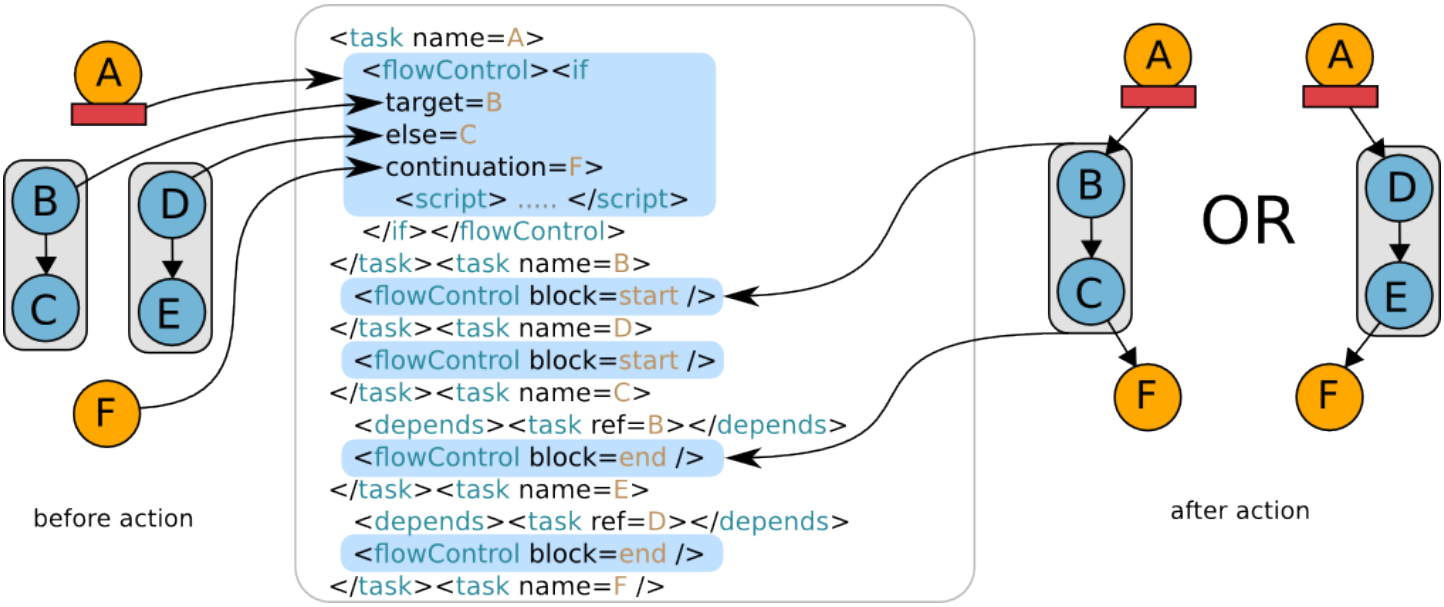


Figure 2.5. If control flow action

- There is no explicit dependency between the initiator and the **if/else** targets. These are **optional links** (ie. A -> B or E -> F in Figure Figure 2.5, “If control flow action”) defined in the **if** task.
- The **if** and **else** flows can be merged by a **continuation** task referenced in the **if** task, playing the role of an **endif** construct. After the branching task, the flow will either be that of the **if** or the **else** task, but it will be continued by the **continuation** task.
- **If** and **else** targets are executed **exclusively**. The initiator however can be the dependency of other tasks, which will be executed normally along the **if** or the **else** target.
- A **task block** can be defined across **if**, **else** and **continuation** links, and not just plain dependencies (i.e. with **A** as **start** and **F** as **end** in Figure 2.5, “If control flow action”).
- If using no continuation task, the if and else targets, along with their children, must be strictly distinct.
- If using a continuation task, the if and else targets must be strictly distinct and valid task blocks.
- **if**, **else** and **continuation** tasks (B, D and F in Figure 2.5, “If control flow action”) cannot have an explicit dependency.
- **if**, **else** and **continuation** tasks cannot be entry points for the job, they must be triggered by the **if** control flow action.
- A task can be target of only one **if** or **else** action. A **continuation** task can not merge two different **if** actions.

2.4.2.4. Looping

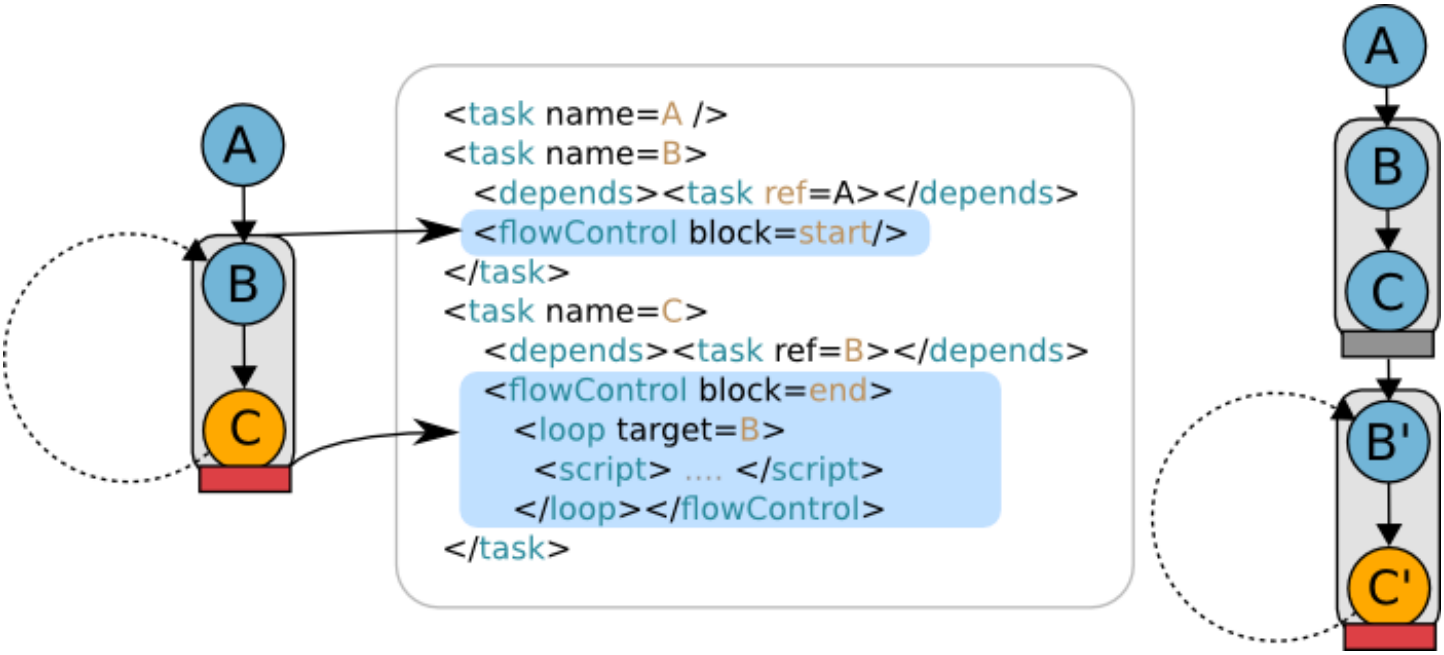


Figure 2.6. Loop control flow action

- The target of a **loop** action must be a parent of the initiator following the dependency chain; this action goes back to a previously executed task.
- Every task is executed at least once; **loop** operates in a **do...while** fashion.
- The target of a **loop** should have only one explicit dependency. It will have different parameters (dependencies) depending if it is executed for the first time or not. The cardinality should stay the same.
- The **loop** scope should be a **task block**: the target is a **start block** task, and the initiator its related **end block task**.

2.4.3. Create a Workflow enabled job

The following section demonstrates the combined use of task replication, branching and looping.

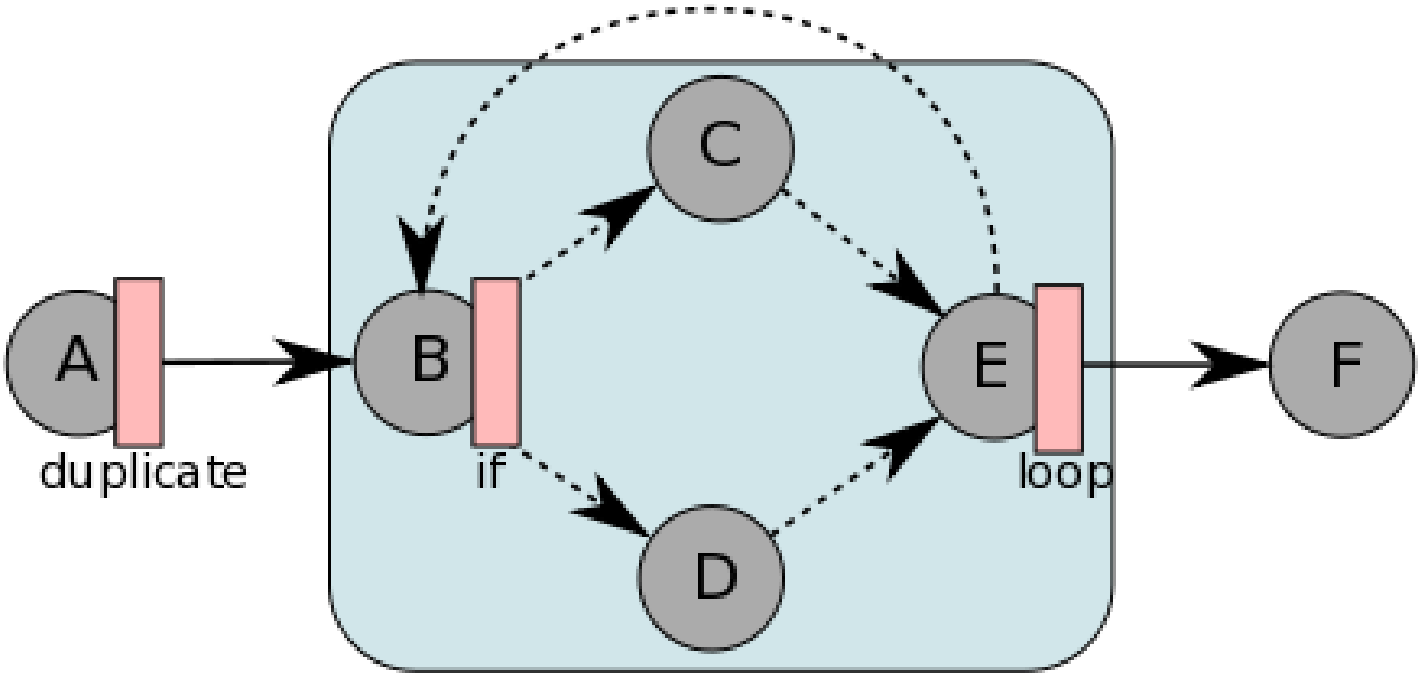


Figure 2.7. A job combining all control flow actions

This job can express the following algorithm:

```
for i in parallel:
  var result
  do:
    if condition:
      result := f1 (result, i)
    else
      result := f2 (result, i)
    endif
  while result > epsilon
done
```

In Figure 2.7, “A job combining all control flow actions”), it can be noted that:

- Task A performs a **replicate** control flow action. This means all tasks that depend on task A will be replicated.
- Task B will be replicated, as it depends on task A, but so will tasks C, D and E as they are included in the same **task block**. Tasks B and E are resepectively the start and end block tasks.
- Task B performs an **if** action. The **if** and **else** targets are tasks C and D, and a **continuation** is performed on task E. An if with continuation operates on task blocks targets, which tasks C and D are, being single tasks.
- Task E performs a **loop** action with task B as target. This action is valid as tasks B (the target) and E (the initiator) form a task block.

2.4.3.1. Task blocks

One important notion that can be extracted from the specification detailed in Section 2.4.2, “Specification” is the existence of **Task Blocks**.

Task blocks are defined by pairs of **start** and **end** tags.

- Each task of the flow can be tagged either **start** or **end**
- Tags can be nested
- Each **start** tag needs to match a distinct **end** tag

Task blocks are very similar to the parenthesis of most programming languages: anonymous and nested start/end tags. The only difference is that a parenthesis is a syntactical information, whereas task blocks are semantic. All combinations of task blocks are syntactically valid, but only those allowed by Section 2.4.2, “Specification” are semantically valid and can be submitted to the scheduler.

The role of Task Blocks is to restrain the expressiveness of the system so that a workflow can be statically checked and validated. A treatment that can be looped or iterated will be isolated in a well-defined task block.

- A **loop** flow action only applies on a Task block: the initiator of the loop must be the end of the block, and the target must be the beginning.
- When using a **continuation** task in an **if** flow action, the **if** and **else** branches must be task blocks.
- If the child of a **replicate** task is a task block, the whole block will be replicated and not only the child of the initiator.

2.4.3.2. Control Flow Scripts

To perform a control flow action such as if, replicate or loop, a **Control Flow Script** is executed on the execution node. This script takes the result of the task as input; meaning a Java object if it was a Java task, or nothing if it was a native task.

The script is executed on the compute node, just after the task's executable. If the executable is a JavaExecutable and returns a result, the variable **result** will be set in the script's environment so that dynamic decisions can be taken with the task's result as input. Native Java objects can be used in a Javascript script using the following syntax:

```
importPackage(java.io);
// the result variable is the TaskResult.value() if it exists
var resFile = new File("/path/to/data/" + new java.lang.String(result));
if (! resFile.exists()) {
  loop = false;
} else {
  loop = true;
  var input = new BufferedReader(new FileReader(f));
  // this variable will determine the number of parallel runs
  runs = java.lang.Integer.parseInt(input.readLine());
  input.close()
}
```

Similarly to how parameters are passed through the **result** variable to the script, the script needs to define variables specific to each action to determine what action the script will lead to.

- A **replicate** control flow action needs to define how many parallel runs will be executed, by defining the variable **runs**:

  ```
  // assuming result is a java.lang.Integer
  runs = result % 4 + 1;
  ```

  The assigned value needs be a strictly positive integer.
- An **if** control flow action needs to determine whether the if or the else branch is selected, it does this by defining the boolean variable **branch**:

  ```
  // assuming result is a java.lang.Integer
  if (result % 2) {
    branch = "if";
  ```

```
} else {
  branch = "else";
}
```

The assigned value needs to be the string value "if" or "else".

- The **loop** control flow action requires setting the **loop**, which will determine wheter looping to the statically defined target is performed, or if the normal flow is executed as would a continue instruction do in a programming language:

```
loop = new java.lang.Boolean(result);
```

The assigned valuee needs to be a boolean.

Failure to set the required variables or the provide a write a valid control flow script will not be treated gracefully and will result in the failure of the task.

2.4.3.3. Using an XML descriptor

The following job is an XML implementation of the job described in Figure 2.7, “A job combining all control flow actions”).

```
<?xml version="1.0" encoding="UTF-8"?>
<job>
 <taskFlow>
  <task name="split">
   <javaExecutable class="/path/to/class/Split" />
   <controlFlow>
    <replicate>
     <script><file path="/path/to/script/duplicate.js" /></script>
    </replicate>
   </controlFlow>
  </task>

  <task name="processing_1">
   <depends><task ref="split" /></depends>
   <javaExecutable class="/path/to/class/Processing_1" />
   <controlFlow block="start">
    <if target="processing_2_a" else="processing_2_b" join="processing_3">
     <script><file path="/path/to/script/if.js" /></script>
    </if>
   </controlFlow>
  </task>

  <task name="processing_2_a">
   <javaExecutable class="/path/to/class/Processing_1" />
   <controlFlow block="start" />
  </task>

  <task name="processing_2_b">
   <javaExecutable class="/path/to/class/Processing_1" />
   <controlFlow block="start" />
  </task>

  <task name="processing_3">
   <javaExecutable class="/path/to/class/Processing_2" />
   <controlFlow block="end">
```

```
      <loop target="processing_1">
        <script><file path="/path/to/script/loop.js" /></script>
      </loop>
    </flow>
  </task>

  <task name="join">
    <depends><task ref="processing_3" /></depends>
    <javaExecutable class="/path/to/class/Join" />
  </task>
 </taskFlow>
</job>
```

2.4.3.4. Using the Java API

This section takes the same example as in Figure 2.7, “A job combining all control flow actions” but focuses on using exclusively the Java API. The code is as follows:

```
TaskFlowJob job = new TaskFlowJob();

JavaTask split = new JavaTask();
split.setName("split");
split.setExecutableClassName("net.example.Split");
FlowScript dup = FlowScript.createReplicateFlowScript(dupScriptContent);
split.setFlowScript(dup);

JavaTask processing_1 = new JavaTask();
processing_1.setName("processing_1");
processing_1.addDependence(split);
processing_1.setExecutableClassName("net.example.Processing_1");
processing_1.setFlowBlock(FlowBlock.START);
FlowScript ifScript = FlowScript.createIfFlowScript(ifScriptContent,
  processing_2_a.getName(), processing_2_b.getName(), processing_3.getName());
processing_1.setFlowScript(ifScript);

JavaTask processing_2_a = new JavaTask();
processing_2_a.setName("processing_2_a");
processing_2_a.setExecutableClassName("net.example.Processing_2_a");

JavaTask processing_2_b = new JavaTask();
processing_2_b.setName("processing_2_b");
processing_2_b.setExecutableClassName("net.example.Processing_2_b");

JavaTask processing_3 = new JavaTask();
processing_3.setName("processing_2");
processing_3.addDependence(processing_1);
processing_3.setExecutableClassName("net.example.Processing_2");
processing_3.setFlowBlock(FlowBlock.END);
FlowScript loop = FlowScript.createLoopFlowScript(loopScriptContent, processing_1.getName());
processing_3.setFlowScript(loop);

JavaTask join = new JavaTask();
join.setName("join");
```

```
join.addDependence(processing_3);
join.setExecutableClassName("net.example.Join");
```

The most relevant method calls are the one used to tag tasks as **start** or **end block**, as well as the ones used to set the Control Flow Script on tasks.

To see how the Control Flow operations can be used and composed within a job, refer to Section 2.4.2, “Specification”

2.4.4. Iteration and replication awareness

When Control Flow actions such as **replicate** or **loop** are performed, some tasks are replicated. To be able to identify replicated tasks uniquely, each replicated task has an **iteration index** and **replication index**.

2.4.4.1. Task names

First, those indexes are reflected inside the names of the tasks themselves. Indeed, task names must be unique inside a job. The indexes are added to the original task name as a suffix, separated by a special character.

- If a task named "T" is replicated after a **loop** action, the newly created tasks will be named "T#1", "T#2", etc. The number following the **#** separator character represents the **iteration index**.
- The same scheme is used upon **replicate** actions: newly created tasks are named "T*1", "T*2", and so on. The separator character *****, and the following number is the **replication index**.
- When combining both of the above, the resulting task names are of the form: "T#1*1", "T#2*4", etc., in that precise order.

Note that these indexes can be safely extracted using the API :

```
JobResult jobResult = scheduler.getJobResult(jobId);
for (Entry<String, TaskResult> result : res.getAllResults().entrySet()) {
  int it = result.getValue().getTaskId().getIterationIndex();
  int dup = result.getValue().getTaskid().getReplicationIndex();
  ...
}
```

2.4.4.2. Iteration and replication indexes in task description

Those indexes are also available when creating a task. They can be obtained using a special string that will be substituted with the actual iteration or replication index at runtime: **$IT** for the iteration index, and **$REP** for the replication index. Those macro can be used in the following locations:

- **Java Executable parameters:**

```
<javaExecutable class="net.example.Executable">
 <parameters>
  <parameter name="images" value="/some/path/images/pic_$IT_$REP.jpg" />
```

```
JavaTask t = new JavaTask();
t.setExecutableClassName("net.example.Executable");
t.addArgument("images","/some/path/images/pic_$IT_$REP.jpg");
```

- **Native Executable arguments:**

```
<staticCommand value="/path/to/bin.sh">
 <arguments>
  <argument value="/some/path/$IT/$REP.dat" />
```

```
NativeTask nt = new NativeTask();
nt.setCommandLine(new String[] { "/path/to/bin.sh", "/path/to/$ID/$REP.dat" });
```

- **Dataspace input and output:**

```
<task name="t1" retries="2">
 <inputFiles>
  <files includes="foo_$IT_$REP.dat" accessMode="transferFromInputSpace"/>
 </inputFiles><outputFiles>
  <files includes="bar_$IT_$REP.res" accessMode="transferToOutputSpace"/>
```

```
task.addInputFiles("foo_$IT_$REP.dat",OutputAccessMode.TransferToOutputSpace);
task.addOutputFiles("bar_$IT_$REP.res",OutputAccessMode.none);
```

- **Script :**

```
<task>
 <pre><script><code language="javascript">
      var f = new java.io.File(".lock_$IT_$REP");
      f.createNewFile();
 </code></script></pre>
```

```
task.setPreScript(new SimpleScript("var f = new java.io.File(\".lock_$ID_$REP\");
 f.createNewFile();", "javascript"));
```

Scripts affected by the macro substitution are: Pre, Post, Command generation and Control Flow. **No substitution will occur in selection scripts.**

2.4.4.3. Iteration and replication indexes in executables

The iteration and replication indexes are available inside the executables launched by tasks.

In Java tasks, the indexes are exported through the following Java properties: **pas.task.iteration** and **pas.task.replication**.

```
public Serializable execute(TaskResult... results) throws Throwable {
  int it  = System.getProperty("pas.task.iteration");
  int dup = System.getProperty("pas.task.replication");
}
```

In a similar fashion, environment variables are set when launching a native executable: **PAS_TASK_ITERATION** and **PAS_TASK_REPLICATION**:

```
#!/bin/sh
/path/to/bin.sh /path/to/file/${PAS_TASK_ITERATION}/${PAS_TASK_REPLICATION}.dat
```

2.5. Defining a Topology for Multi-Nodes Tasks

If a multi-nodes task is highly communicative and you would like to execute it on the set of the closest nodes or if it has other restrictions (i.e. has to be executed exclusively on host) the topology parameter can be specified. The topology in our case takes into account only

network latencies between nodes and does not imply bandwidth or other parameters. Using networks latencies you may ask for you tasks the set of nodes within some latency threshold, nodes on the same host and many more.

The definition of the topology is quite straightforward. First we need to define the **parallel** environment element:

```
<task name="MultiNodesTask">
  <description>Will control the workers in order to find the prime number</description>
  <parallel numberOfNodes="4"/>
  <javaExecutable
    class="org.ow2.proactive.scheduler.examples.MultiNodeExample">
    <parameters>
      <parameter name="numberToFind" value="100"/>
    </parameters>
  </javaExecutable>
</task>
```

Here we defined the task required 4 nodes to be executed. The number of nodes must be greater than 1 for any multi-nodes task.

2.5.1. Topology types

Then we define one of the possible topology types:

- **Arbitrary** topology does not imply any restrictions on nodes location

```
<parallel numberOfNodes="4">
  <topology>
    <arbitrary/>
  </topology>
</parallel>
```

- **Best proximity** - the set of closest nodes among those which are not executing other tasks.

```
<parallel numberOfNodes="4">
  <topology>
    <bestProximity/>
  </topology>
</parallel>
```

- **Threshold proximity** - the set of nodes within a threshold proximity (in microseconds).

```
<parallel numberOfNodes="4">
  <topology>
    <thresholdProximity threshold="100"/>
  </topology>
</parallel>
```

- **Single Host** - the set of nodes on a single host.

```
<parallel numberOfNodes="4">
  <topology>
    <singleHost/>
  </topology>
</parallel>
```

- **Single Host Exclusive** - the set of nodes of a single host exclusively. The host with selected nodes will be reserved for the user.

```
<parallel numberOfNodes="4">
  <topology>
    <singleHostExclusive/>
```

```
  </topology>
</parallel>
```

For this task the scheduler will try to find a host with 4 nodes. If there is no such a host another one will be used with a bigger capacity (if exists). In this case extra nodes on this host will be also occupied by the task but will not be used for the execution.

- **Multiple Hosts Exclusive** - the set of nodes filled in multiple hosts. Hosts with selected nodes will be reserved for the user.

```
<parallel numberOfNodes="4">
  <topology>
    <multipleHostsExclusive/>
  </topology>
</parallel>
```

For this task the scheduler will try to find 4 nodes on a set of hosts. If there is no such a set which gives you exactly 4 nodes another one will be used with a bigger capacity (if exists). In this case extra nodes on this host will be also occupied by the task but will not be used for the execution.

- **Different Hosts Exclusive** - the set of nodes each from a different host. All hosts with selected nodes will be reserved for the user.

```
<parallel numberOfNodes="4">
  <topology>
    <differentHostsExclusive/>
  </topology>
</parallel>
```

For this task the scheduler will try to find 4 hosts with one node on each. If number of "single node" hosts are not enough hosts with bigger capacity will be selected. In this case extra nodes on this host will be also occupied by the task but will not be used for the execution.

Warning

You should consider the topology as an optimization for multi-nodes task execution and use it with moderation. Note that using the topology will delay you task to be scheduled (especially when it is used in combination with selection scripts) as the resources appropriate for you task will be harder to find. Even though the execution time of the task should reduce the scheduling time will definitely increase.

2.5.2. Setting up a topology using Java API

We already know how to create the task using Java API of the scheduler (see Section 2.2.1.3, “Create and add a Java task using the Java API”). The topology could be also specified as follows:

```
// setting up the parallel environment with best proximity descriptor
task.setParallelEnvironment(new ParallelEnvironment(5, TopologyDescriptor.BEST_PROXIMITY));
// or the parallel environment with 100 microseconds threshold proximity
task.setParallelEnvironment(new ParallelEnvironment(5, new ThresholdProximityDescriptor(100)));
// or 5 nodes all on the same host
task.setParallelEnvironment(new ParallelEnvironment(5, TopologyDescriptor.SINGLE_HOST));
// or 5 nodes all on the same host exclusively (on other tasks will be executed there in the same time)
task.setParallelEnvironment(new ParallelEnvironment(5, TopologyDescriptor.SINGLE_HOST_EXCLUSIVE));
// or 5 nodes from several hosts exclusively (on other tasks will be executed there in the same time)
task.setParallelEnvironment(new ParallelEnvironment(5,
   TopologyDescriptor.MULTIPLE_HOSTS_EXCLUSIVE));
// or 5 hosts (1 node from each host exclusively)
task.setParallelEnvironment(new ParallelEnvironment(5,
   TopologyDescriptor.DIFFERENT_HOSTS_EXCLUSIVE));
```

In order to select nodes according to the topology requirement the scheduler ask for nodes from the resource manager using this criterion. The algorithms using by resource manager behind the scene could be found in the Resourcing documentation[1].

2.6. Submit a job to the Scheduler

The submission will perform some verifications to ensure that a job is correctly formed. Then, the job is inserted in the pending queue and waits 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 to the queue of finished jobs and will wait until the user retrieves his result. Their are three ways to submit a job to the Scheduler described in the following sub-sections.

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

To submit a job using the graphical tools, you have first to create a job XML Descriptor. Once done, please refer to Chapter 5, *ProActive Scheduler Eclipse Plugin*.

2.6.2. Submit a job using the shell command

Use the provided shell script **scheduler-client[.bat]** to submit a job using command line. This script (bin/[os]/scheduler-client[.bat] used to submit a job) has 1 mandatory option and 2 optional :

- **Job to schedule** (mandatory) - It can be an XML Job descriptor, a simple flat file, or a path to a native command to launch : Use -submit followed by the path or the command to give, and :
  - To schedule an XML job file: use -submit followed by the path of an XML descriptor
  - To schedule a flat file containing native commands: use -submit followed the by the path of a flat file containing native commands and -cmdf to specify that it is a command file.
  - To schedule a path to a native command : use -submit followed by the path of the native command to launch, with its launching arguments. Here, it submits a job made of one native task. Then, use -cmd to specify that it is a command path.

  For -cmd and -cmdf options, you can use aditional parameters. For instance, you can specify a log file that will contain all STDOUT and STDERR of job execution, with -o followed by the path of a file. This file is created (emptied if exits) at each beginning of job execution. You can specify a selection script with -s, followed by the path of a selection script file. You can specify a job name, with -jn followed by a string specifying a name. These three options (-s, -o and -jn) are not taken into account if -cmd or -cmdf are not specified option, because job name, selection scripts and log file are directly specified in the XML job descriptor.
- **The URL of a started scheduler** (optional) - To use this option, add a "-u URL" argument. If not mentioned, the script will connect to an existing local Scheduler.
- **Your login** (optional) - To use this option, add a "-l login" argument . If you use this option, only your password will be requested. Otherwise, both will be.

Examples:

scheduler-client[.bat] -submit ../../samples/jobs_descriptors/Job_with_dep.xml -l login -u //localhost:1099/ will submit the Job_with_dep job to a local ProActive Scheduler on port 1099 with the username 'login', and only your password will be requested. Authorized username and password are defined by the administrator.

scheduler-client[.bat] -submit ../../samples/jobs_descriptors/job_native_linux_cmd_file/cmds_file.cmd -cmdf -l demo //localhost/ -jn myJob -o myJob.log will submit a job defined in a flat file, containing 4 native commands. Job is named "myJob", and a log file named "myJob.log" will be created.

scheduler-client[.bat] -submit ../../samples/jobs_descriptors/job_native_linux_cmd_file/myCmd.sh 4 -cmd -l demo -u //localhost -s ../../samples/scripts/selection/select.rb will submit the native command "myCmd.sh" with "4" as the only argument. A selection script is associated for this native command: "select.rb".

For more information, use -h (or --help) option (i.e. "scheduler-client[.bat] -h").

[1] http://proactive.inria.fr/trunk/Resourcing/multiple_html/index.html

2.6.3. Submit a job using the Java API

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

```
//join an existing ProActive Scheduler retrieving an authentication interface.
try {
   SchedulerAuthenticationInterface auth = SchedulerConnection.waitAndJoin("protocol://host:port");
   //connect and log to the Scheduler. Valid username and password are defined by the administrator
   Scheduler scheduler = null;
   try {
      // (1) preferred authentication method
      scheduler = auth.login(Credentials.getCredentials());
   } catch (KeyException ke) {
      try {
         // (2) alternative authentication method
         PublicKey pubKey = auth.getPublicKey();
         if (pubKey == null) {
             pubKey = Credentials.getPublicKey(Credentials.getPubKeyPath());
         }
         scheduler = auth.login(Credentials.createCredentials(new CredData("demo", "demo"), pubKey));
      } catch (KeyException ke2) {
         //cannot find public key !
      }
   }
   // submitting a new job and get the associated id
   JobId myJobId = scheduler.submit(job);
} catch (ConnectionException e) {
   //cannot join scheduler !
   e.printStackTrace();
}
```

Connecting to the Scheduler implicates using a keypair infrastructure to establish a secure channel on which the credentials (username and password) will be sent securely. There are two ways to connect to the Scheduler, as described in the example above:

1. Retreive the Credentials from disk: supposes the script **create-cred[.bat]** was previously used to generate those credentials. It requires knowing the location of the public key corresponding to the private key that will be used for decryption on server side. To obtain the key, you can either have it on your local drive, or ask the remote Scheduler:

   ```
   # use local public key to generate credentials for user 'demo', password will be specified in interactive mode
   /bin/unix $ ./create-cred -F $HOME/.proactive/scheduler_pubkey -o $HOME/.proactive/my_encrypted_credentials -l
    demo
   # use rmi://example.com:1099/'s public key to generate credentials for user 'admin' with pass 'admin' in non-
   interactive mode
   /bin/unix $ ./create-cred -S rmi://example.com:1099/ -o $HOME/.proactive/my_encrypted_credentials -l admin -p
    admin
   # also store user ssh private key in credentials to be able to execute task under user ID
   /bin/unix $ ./create-cred -S rmi://example.com:1099/ -o $HOME/.proactive/my_encrypted_credentials -l admin -p
    admin -k $HOME/.ssh/id_rsa
   ```

   Credentials can now be retreived when properly designated by the **pa.common.auth.credentials** property; ie using **java -Dpa.common.auth.credentials=$HOME/.proactive/my_encrypted_credentials**. Please type create-cred -h to have the complete list of possible options.

2. Create the encrypted Credentials on client side: as safe as the previous method, but requires user input, which prevents automation, or storing clear credentials on the disk, which can result to security breaches. This method also requires knowing the public key, which should be offered by the Scheduler through the Authentication object with the getPubKey() method. If the Scheduler doesn't know the public key, it can be stored locally on client side and designated by the **pa.common.auth.pubkey** Java property; ie using **java -Dpa.common.auth.pubkey=$HOME/.proactive/pub.key**.

In any cases, you can create your own encrypted credentials using login/password and an optional ssh private key. The ssh private key (if specified) will only be used if the task property 'runAsMe' is true and if the node is configured to accept ssh key authentication.

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 information on this job.

If you want to get a job from a XML job file descriptor, just add the following line before submitting :

```
//join an existing ProActive Scheduler as below
...
//create the job, 'filePath' is the path of the XML job descriptor
Job job = JobFactory.getFactory().createJob(filePath);
//then submit it
scheduler.submit(job);
```

2.7. Get a job result

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

2.7.1. 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 5, *ProActive Scheduler Eclipse Plugin*.

2.7.2. Get a job result using the shell command

To get the result of you job using a command line, launch the command **scheduler-client[.bat] -jobresult jobID** script in the bin/[os]/ directory. This script has 2 optional options:

- The URL of a started Scheduler ("-u URL" option). If you don't use this, it will try to connect to a started scheduler on the local host.
- Your login ("-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 information, use -h (or --help) option (i.e. "scheduler-client[.bat] -h").

2.7.3. Get a job result using the Java API

To do it in Java, use the **getJobResult(JobId)** method in the **Scheduler** interface and the job ID you got when you submitted it. 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.
- A precious result list that contains every result or exception returned by the task marked as precious.

This result will be given to you exactly like you returned it in your executable. To know when a job 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 Scheduler interface
Scheduler scheduler = auth.login(Credentials.getCredentials());
// get the result of the job
JobResult myResult = scheduler.getJobResult(myJobId);
//look at inside the JobResult to retrieve TaskResult...
```

Note : The above example is useful when you kept a reference of myJobId. If you want to submit a job and disconnect, just keep the id as a string (using myJobId.value()). Then when reconnecting the Scheduler to get the result, just use the getJobResult(String jobId) method.

2.8. 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... To add a listener, create your listener implementing the **SchedulerEventListener** interface and add it to the scheduler. You will then receive the scheduler state containing some information about the current scheduling state. See the ProActive Scheduler JAVADOC for more details.

It is also possible to specify if you want to receive events concerning your jobs, or any job from any user in the Scheduler.

```
//make your listener
SchedulerEventListener mySchedulerEventListener = new SchedulerEventListener () {
   jobStateUpdatedEvent(NotificationData<JobInfo> notification){
      switch(notification.getEventType()){
         case JOB_RUNNING_TO_FINISHED :
            //if my job is finished
            if (notification.getData().getJobId().equals(myJobId)){
               //get its result
               JobResult myResult = scheduler.getJobResult(myJobId);
            }
            break;
      }
   }
   //Implement other methods...
}
//add the listener to the scheduler specified which events you want to receive.
scheduler.addEventListener(mySchedulerEventListener, true, SchedulerEvent.JOB_RUNNING_TO_FINISHED);
```

This example shows you how to listen to the scheduler events (only finished job event in the previous example). Yet, you can listen for every event you want contained in this interface.

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

2.9. Using the Scheduler controller

The Scheduler controller provides a way to interact with the Scheduler. Start it with the **bin/[os]/scheduler-client[.bat]** script. You can use it as a command line (as shown in some examples above, go to Section 2.9.1, “Command line mode”) or in interactive mode (go to Section 2.9.2, “Interactive mode”).

2.9.1. Command line mode

Here is the displayed help when using **scheduler-client[.bat] -h** command:

```
usage: scheduler-client [-c <arg>] [-cmd] [-cmdf] [-env <filePath> | -freeze | -jo <jobId> | -jp <jobId newPriority> | -jr <jobId> |
       -js <jobId> | -kill | -kj <jobId> | -lj | -lrm <rmURL> | -ma | -p <fullName> | -pause | -pj <jobId> | -resume | -rj <jobId> | -rmj
       <jobId> | -rp | -s <XMLDescriptor> | -sf <filePath arg1=val1 arg2=val2 ...> | -shutdown | -start | -stats | -stop | -test | -to <jobId
       taskName> | -tr <jobId taskName> | -ua <username>]  [-h] [-jn <jobName>]     [-k <sshkeyFilePath>]   [-l <login>]    [-o <logFile>]
       [-ss <selectScript>]       [-u <schedulerURL>]
 -c,--credentials <arg>                              Path to the credentials (/home/jlscheef/.proactive/security/creds.enc).
 -cmd,--command                                      <ctl> If mentionned, -submit argument becomes a command line, ie: -submit command
                                                     args...
 -cmdf,--commandf                                    <ctl> If mentionned, -submit argument becomes a text file path containing command
                                                     lines to schedule
 -env,--environment <filePath>                       Execute the given script as an environment for the interactive mode
 -freeze,--schedulerfreeze                           <ctl> Freeze the Scheduler (cause all non-running tasks to be paused)
 -h,--help                                           Display this help
 -jn,--jobname <jobName>                             <ctl> Used with -cmd or -cmdf, specify the job name
 -jo,--joboutput <jobId>                             <ctl> Get the output of the given job
 -jp,--jobpriority <jobId newPriority>               <ctl> Change the priority of the given job (Idle, Lowest, Low, Normal, High,
                                                     Highest)
 -jr,--jobresult <jobId>                             <ctl> Get the result of the given job
 -js,--jobstate <jobId>                              <ctl> Get the current state of the given job (Also tasks description)
 -k,--key <sshkeyFilePath>                           (Optional) The path to a private SSH key
 -kill,--schedulerkill                               <ctl> Kill the Scheduler
 -kj,--killjob <jobId>                               <ctl> Kill the given job (cause the job to finish)
 -l,--login <login>                                  The username to join the Scheduler
 -lj,--listjobs                                      <ctl> Display the list of jobs managed by the scheduler
 -lrm,--linkrm <rmURL>                               <ctl> Reconnect a RM to the scheduler
 -ma,--myaccount                                     <ctl> Display current user account information
 -o,--output <logFile>                               <ctl> Used with submit action, specify a log file path to store job output
 -p,--policy <fullName>                              <ctl> Change the current scheduling policy
 -pause,--schedulerpause                             <ctl> Pause the Scheduler (cause all non-running jobs to be paused)
 -pj,--pausejob <jobId>                              <ctl> Pause the given job (pause every non-running tasks)
 -resume,--schedulerresume                           <ctl> Resume the Scheduler
 -rj,--resumejob <jobId>                             <ctl> Resume the given job (restart every paused tasks)
 -rmj,--removejob <jobId>                            <ctl> Remove the given job
 -rp,--reloadpermissions                             <ctl> Reloads the permission file
 -s,--submit <XMLDescriptor>                         <ctl> Submit the given job XML file
 -sf,--script <filePath arg1=val1 arg2=val2 ...>     <ctl> Execute the given javascript file with optional arguments.
 -shutdown,--schedulershutdown                       <ctl> Shutdown the Scheduler
 -ss,--selectscript <selectScript>                   <ctl> Used with -cmd or -cmdf, specify a selection script
 -start,--schedulerstart                             <ctl> Start the Scheduler
 -stats,--statistics                                 <ctl> Display some statistics about the Scheduler
 -stop,--schedulerstop                               <ctl> Stop the Scheduler
 -test                                               <ctl> Test if the Scheduler is successfully started by committing some examples
 -to,--taskoutput <jobId taskName>                   <ctl> Get the output of the given task
 -tr,--taskresult <jobId taskName>                   <ctl> Get the result of the given task
 -u,--url <schedulerURL>                             The scheduler URL (default rmi://localhost/)
 -ua,--useraccount <username>                        <ctl> Display account information by username

NOTE : if no <ctl> command is specified, the controller will start in interactive mode.
```

Figure 2.8. Scheduler controller help

To use the controller as a command line, just specify at least one 'ctl' option. Let's illustrate that with some examples:

- **scheduler-client[.bat] -l demo -submit path/to/job/xml/descriptor**: tries to connect user demo, prompts for its password and submits the given XML descriptor file.
- **scheduler-client[.bat] -l demo -cmd -jn test -submit ls -al**: tries to connect user demo, prompts for its password and submits the given command line in a job named 'test'. In this case, the *-cmd* option forces the argument of *-submit* option to be a command line.
- **scheduler-client[.bat] -cmdf -s path/to/a/selection/script -submit path/to/a/text/file**: prompts for a username and its password and submits the given commands file. It also includes the given selection script file. In this case, the *-cmdf* option forces the argument of *-submit* option to be a flat text file that contains commands (One command per line, each line will be a task).
- **scheduler-client[.bat] -joboutput 12**: prompts for a username and its password and displays the output written by job 12 if it is finished.
- **scheduler-client[.bat] -jobresult 34**: prompts for a username and its password and displays the result of job 34 if it is finished.
- **scheduler-client[.bat] -priority 56 Lowest**: prompts for a username and its password and requests the Scheduler to change job 56 priority to 'LOWEST'.

To use the controller, you can either type your login and password in interactive mode, or use the **--use-cred** option to attempt retreiving encrypted credentials from disk. In both cases, the credentials need to be encrypted when transiting on the network: you need to have the Scheduler's public key, corresponding to the private key that will be used to decrypt the credentials.

1. In interactive mode, this public key will be asked by the controller to the remote Scheduler it is attempting to contact: if the Scheduler does not know it or is unable to forward it to the controller, is must be specified through the **pa.common.auth.pubkey property**, ie. using **java -Dpa.common.auth.pubkey=$HOME/.proactive/my_encrypted_credentials**.

2. In batch mode, this public key will be used when encrypting the credentials with the bin/[os]/createCred[.bat] script:

```
# use local public key to generate credentials for user 'demo'
/bin/unix $ ./create-cred -F $HOME/.proactive/scheduler_pubkey -o $HOME/.proactive/my_encrypted_credentials -l
 demo
# use rmi://example.com/'s public key to generate credentials for user 'admin' with pass 'admin' in non-interactive
 mode
/bin/unix $ ./create-cred -S rmi://example.com/ -o $HOME/.proactive/my_encrypted_credentials -l admin -p admin
```

When running the controller, Credentials will now be retreived when properly designated by the **pa.common.auth.credentials** property; ie. using **java -Dpa.common.auth.pubkey=$HOME/.proactive/my_encrypted_credentials**, or when using the **--credentials PATH** option.

Once the command launched, it will terminate by displaying a successful message or the cause of a possible failure as an exception.

2.9.2. Interactive mode

Another way to interact with the Scheduler is using the interactive mode of the controller. To start it in this mode, just execute the bin/[os]/scheduler-client[.bat] command without specifying a 'ctl' option: **scheduler-client[.bat] -l demo -u rmi://localhost:1313/** is an example which connects the controller to a local Scheduler on the port 1313 with 'demo' login.

Once connected, type *?* or *help* to get the list of provided functions:

```
> ?

Scheduler controller commands are :

 submit(XMLdescriptor)     Submit a new job (parameter is a string representing the job XML descriptor URL)
 submitCmd(CommandFilePath,jobName,output,selectionScript) Submit a new job where each task is a line in the commandFile path. Other arguments are
 priority(id,priority)     Change the priority of the given job (parameters are an int or a string representing the jobId AND a string representing
                           Priorities are Idle, Lowest, Low, Normal, High, Highest
 pausejob(id)              Pause the given job (parameter is an int or a string representing the jobId)
 resumejob(id)             Resume the given job (parameter is an int or a string representing the jobId)
 killjob(id)               Kill the given job (parameter is an int or a string representing the jobId)
 jobresult(id)             Get the result of the given job (parameter is an int or a string representing the jobId)
 taskresult(id,taskName)   Get the result of the given task (parameter is an int or a string representing the jobId, and the task name)
 joboutput(id)             Get the output of the given job (parameter is an int or a string representing the jobId)
 taskoutput(id,taskName)   Get the output of the given task (parameter is an int or a string representing the jobId, and the task name)
 removejob(id)             Remove the given job from the Scheduler (parameter is an int or a string representing the jobId)
 jobstate(id)              Get the current state of the given job (parameter is an int or a string representing the jobId)
 listjobs()                Display the list of jobs managed by the scheduler
 stats()                   Display some statistics about the Scheduler
 myaccount()               Display current user account information
 account(username)         Display account information by username
 reloadpermissions()       Reloads the permission file
 exec(scriptFilePath)      Execute the content of the given script file (parameter is a string representing a script-file path)
 start()                   Start Scheduler
 stop()                    Stop Scheduler
 pause()                   Pause Scheduler, causes every jobs but running one to be paused
 freeze()                  Freeze Scheduler, causes all jobs to be paused (every non-running tasks are paused)
 resume()                  Resume Scheduler, causes all jobs to be resumed
 shutdown()                Wait for running jobs to finish and shutdown Scheduler
 kill()                    Kill every tasks and jobs and shutdown Scheduler
 linkrm(rmURL)             Reconnect a Resource Manager (parameter is a string representing the new rmURL)
 changepolicy(fullName)    Change the current scheduling policy, (argument is the new policy full name)
 reconnect()               Try to reconnect this console to the server
 cnslhelp() or ?c          Displays help about the console functions itself
 exit()                    Exit Scheduler controller

>
```

Figure 2.9. Scheduler controller interactive help

The interactive controller is **based on JavaScript language** where some methods (describe in Figure 2.9, “Scheduler controller interactive help”) have been defined in order to facilitate its usage. Here is an example of what can be done with the console:

```
 > submit("../../samples/jobs_descriptors/Job_PI.xml");
Job successfully submitted ! (id=1)
```

```
 > myId = submit("../../samples/jobs_descriptors/Job_PI.xml");
Job successfully submitted ! (id=2)

 > pausejob(1);
Job 1 paused.

 > resumejob(1);
Job 1 resumed.

 > jobresult(myId);
Job 2 result =>

    Average1 : 3.1414783733333334
    Computation3 : 3.1414446
    Computation4 : 3.14154732
    Average2 : 3.141520226666667
    Computation1 : 3.14146984
    LastAverage : 3.1414993000000004
    Computation5 : 3.14129996
    Computation2 : 3.14152068
    Computation6 : 3.1417134

 > joboutput(myId);
Job 2 output =>

Average1 :
Parameters are :
3.14146984
3.14152068
3.1414446
Average is : 3.1414783733333334

Average2 :
Parameters are :
3.14154732
3.14129996
3.1417134
Average is : 3.141520226666667

LastAverage :
Parameters are :
3.1414783733333334
3.141520226666667
Average is : 3.1414993000000004


 > myResult = jobresult(1)
Job 1 result =>

    Average1 : 3.14164032
    Computation4 : 3.14176412
    Computation3 : 3.14162724
    Average2 : 3.1416647466666667
```

```
    Computation1 : 3.14182172
    LastAverage : 3.1416525333333336
    Computation5 : 3.1416124
    Computation2 : 3.141472
    Computation6 : 3.14161772

  > println(myResult.getName());
job_PI
```

As you can see, the provided methods are useful to make basic actions. It is also possible to get instances from Scheduler API. As shown, the *myResult = result(1)* instruction leads to get an instance of JobResult in the 'myResult' variable. As in Java, use the method of the JobResult API to get other informations. Here is an example:

```
  > prio = myResult.getJobInfo().getPriority();

  > status = myResult.getJobInfo().getStatus();

  > nbfinished =  myResult.getJobInfo().getNumberOfFinishedTasks();

  > duration = myResult.getJobInfo().getFinishedTime()-myResult.getJobInfo().getStartTime();

  > println("priority : "+prio);  println("status : "+status); println("finished tasks : "+nbfinished); println("duration = "+duration
+" ms");
priority : Normal
status : Finished
finished tasks : 9
duration = 94386 ms
```

The *exec(...)* function provides a way to evaluate a javascript file. For example, just insert the code below in a '.js' file with:

```
myr = jobresult(1);
prio = myr.getJobInfo().getPriority();
status = myr.getJobInfo().getStatus();
nbfinished =  myr.getJobInfo().getNumberOfFinishedTasks();
duration = myr.getJobInfo().getFinishedTime()-myr.getJobInfo().getStartTime();
println("priority : "+prio);  println("status : "+status); println("finished tasks : "+nbfinished); println("duration = "+duration+"
 ms");
```

The result will be exactly the same as the one in the code shown in the previous example.

When error occurred in a command, it is possible to show the stack trace on demand. Indeed, the execution stops and prompts the user to show the stackTrace or not. In order to script this interactive controller, you can disable stack trace on demand. To do so, use the *exMode(...)* function. The first parameter defines whether the stack trace will be displayed while the second one defines if the stack trace has to be displayed every time or on demand. Setting the first parameter to 'false' disables the second one.

As an example, let's see the following script:

```
  > exMode(false);
Exception display mode changed : stack trace not displayed
```

```
 > for (i=0;i<5;i++){submit("../../samples/jobs_descriptors/Job_PI.xml")};
Job successfully submitted ! (id=3)
Job successfully submitted ! (id=4)
Job successfully submitted ! (id=5)
Job successfully submitted ! (id=6)
Job successfully submitted ! (id=7)

 > for (i=0;i<5;i++){jobresult(i);}
Error on job 0 : The job represented by this ID is unknow !
Error on job 1 : The job represented by this ID is unknow !
Error on job 2 : The job represented by this ID is unknow !
Job 3 result =>

    Average1 : 3.1416381333333336
    Computation4 : 3.14196316
    Computation3 : 3.14175216
    Average2 : 3.141720133333333
    Computation1 : 3.14169736
    LastAverage : 3.1416791333333336
    Computation5 : 3.14180604
    Computation2 : 3.14146488
    Computation6 : 3.1413912
Job 4 is not finished or unknown !
```

In this case, we get some error messages but no exception and the script can terminate without prompting to view stack trace.

Note : Interactive mode can also be started using a javascript environment. To do so, just put a file named **scheduler-client.js** in the [user.home]/.proactive directory OR just reference your own file when starting the controller :

- **scheduler-client -c -js myenv.js** : This command will start the scheduler controller in interactive mode first loading the 'myenv.js' script and then connecting the scheduler using default credentials path !

Chapter 3. Administration guide

3.1. Scheduler Architecture

3.1.1. Scheduler Global Architecture

The ProActive Scheduler Service is the result of a collaboration between 2 entities (the ProActive Scheduler and the Resource Manager). Each one of them has its own functionality.

The ProActive Scheduler is the main entity and is a non GUI daemon which is connected to the Resources Manager. It is in charge of scheduling submitted jobs in accordance with the scheduling policy.

In order to launch jobs, the ProActive Scheduler has to obtain nodes (resources) from the Resources Manager. As described hereafter, the user interacts only with the ProActive Scheduler entity and the managed resources can be simple hosts or peer to peer resources. For full documentation about the Resource Manager, please refer to its own documentation.

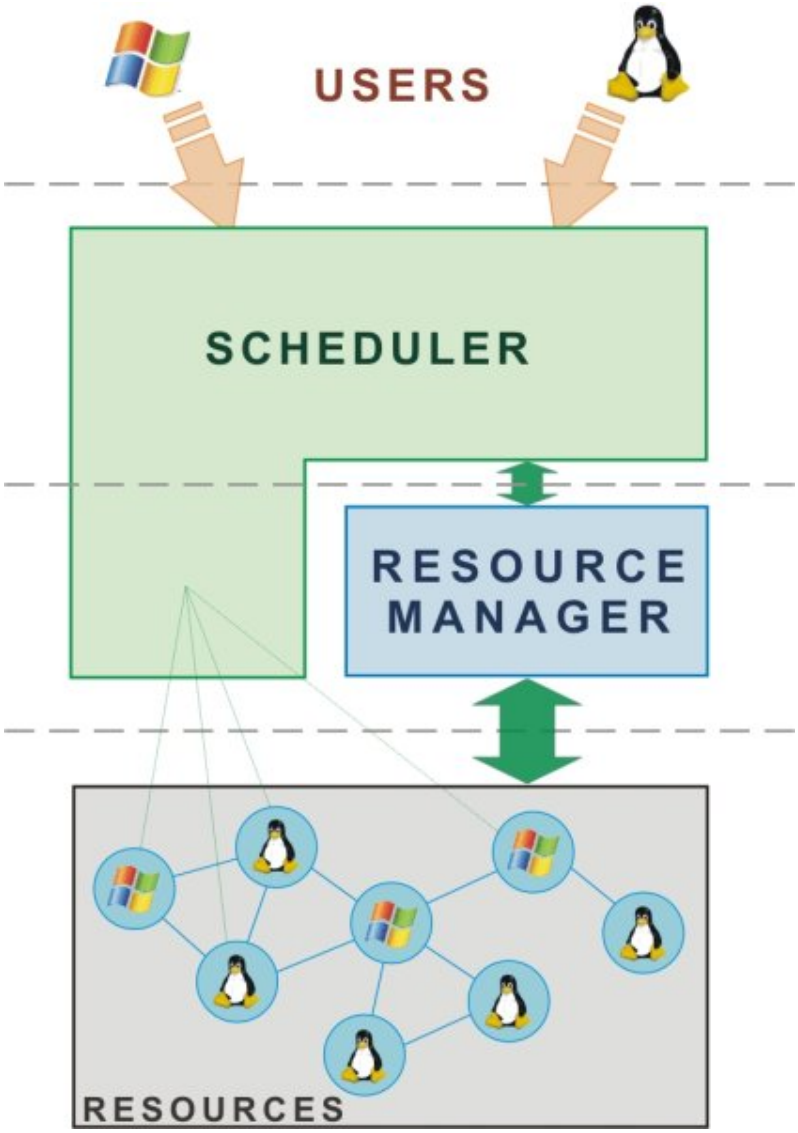


Figure 3.1. The ProActive Scheduler Entities

In this section, we will explain how the **scheduler entity works and how it can be used.**

3.1.2. Scheduler Entity Architecture

The architecture of the ProActive Scheduler (Figure 3.2, "The ProActive Scheduler Entity") is built around 3 Active Objects: To know more about Active Object, please refer to the ProActive Documentation.

- **The Authentication interface** which is the first object that the user may have to contact. It is in charge of authenticating the user and allowing him to access (or not) to the Scheduler. The authentication security system can interact with files or LDAP.
- **The Front-end** which is the interface returned by the Authentication Interface and allows interaction with the ProActive Scheduler. This interface allows users to submit jobs, get scheduling state, retrieves job result...
- **The Core** which is the main entity of the ProActive Scheduler. It is in charge of scheduling Jobs according with the policy (FIFO by default), retrieving scheduling events to the user and making storages.

Users cannot interact directly with the ProActive Scheduler Core. They have to use the Front-end gateway.

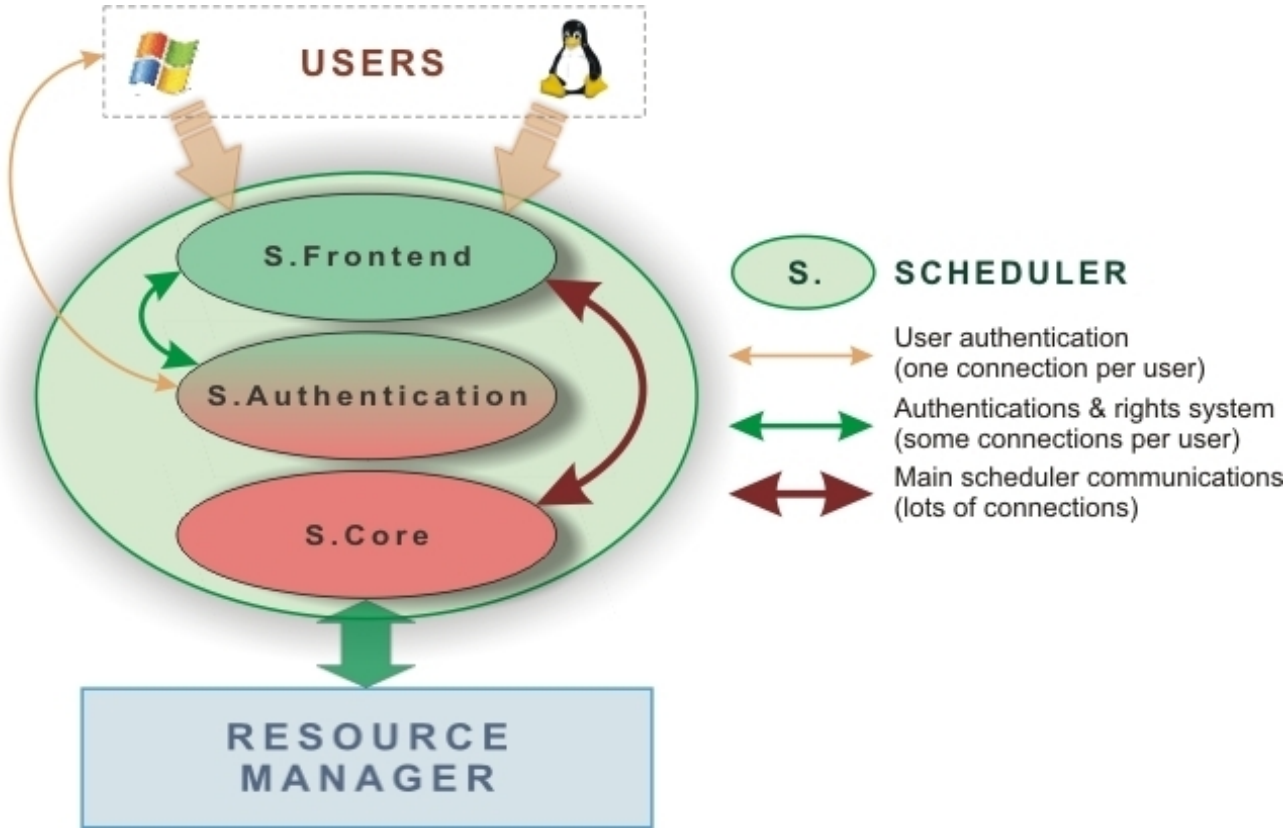


Figure 3.2. The ProActive Scheduler Entity

3.2. Start the ProActive Scheduler

The ProActive Scheduler has to be started with a database that will be used to save scheduling process in case of crash. It allows the ProActive Scheduler to restart with a consistent state. The ProActive Scheduler will either use an existing database or create a new one if it cannot find one. (Supported databases are Java DERBY and MySQL)

- The database get its properties in the **config/scheduler/database/hibernate/hibernate.cfg.xml** configuration file. Here is an example of configuration:

```
<property name="connection.driver_class">org.apache.derby.jdbc.EmbeddedDriver</property>
```

```
<property name="connection.url">jdbc:derby:../../SCHEDULER_DB;create=true</property>
<property name="connection.username">scheduler</property>
<property name="connection.password"></property>
<!-- Derby dialect  -->
<property name="dialect">org.hibernate.dialect.DerbyDialect</property>
```

- **driver_class** defines which driver to use for the current dataBase implementation.
- **url** defines the protocol, the url and the name used for this dataBase. The goal of this two first properties is to provide a way to change the implementation of the database and its directory. To start the Scheduler with the provided implementation, just don't modify them.
- **username** is the user name authorized to create and manage the database ('scheduler' by default)
- **password** is the password of the authorized user (empty by default)
- **dialect** If you intend to change the database implementation, don't forget to change the dialect to match the given implementation.

It is also possible to configure some properties that rely on the Scheduler. The file **settings.ini** in the **config/scheduler** directory contains every property that can be modified by administrator. The **PASchedulerProperties** class provide a way to override this file with your own one. If you do so, every overridden properties will be used. It is not necessary to specify every properties inside your own file. Here's a way to override default PAScheduler properties file by the code:

```
PASchedulerProperties.updateProperties("your_property_file");
```

The last method to override a property is to specify it into the the java command line using the -d option. You just have to name the property as it is done in the properties file, and give it a value. (ex: java -dpa.scheduler.core.nodepingfrequency=30000...). Such a property can be added in the provided scripts (scheduler-start[.bat] for example).

The priority order between two definition of a same property is as follows: the definition with the highest priority is those defined using the Java command line, then it is the one defined in your own properties file and finally, the lowest priority is given to the default definition.

Let's go on to the startup of the ProActiveScheduler. To start it with a shell command, go on to the next section. To start it using the Java API, please refer to Section 3.2.3, “Start the Scheduler using the Java API”.

3.2.1. ProActive Scheduler properties

Note1: some properties require a file path. File path can be given relative, and if so, if will be relative to the Scheduler home directory (i.e. to the application root directory) OR with an absolute path. Also note that each property can be overridden by using the JVM properties when starting the Scheduler. (i.e. scheduler-start[.bat] -Dpa.scheduler.core.timeout=1000)

Note2: the main properties file could be found at 'config/scheduler/settings.ini'

Scheduler properties

- *pa.scheduler.home*: defines Scheduler home directory. The home directory is the one containing the dist, doc, lib, classes, config directory (default is ., the current directory).
- *pa.scheduler.core.timeout*: timeout used in the main scheduling loop. If no new request is received by the SchedulerCore entity, no new resources are available, and tasks are running, it is the time to wait before the next scheduling loop specified in milliseconds (default is 2000ms).
- *pa.scheduler.core.nodepingfrequency*: time interval between each failed node checking specified in second (default is 20s).
- *pa.scheduler.classserver.usecache*: boolean that specifies if the class definitions used in task class servers must be cached or not. Set to false to preserve memory usage in SchedulerCore (default is true).
- *pa.scheduler.classserver.tmpdir*: directory used to store a job classpath. If this value is not set, it uses the default TMP directory (default is not set).

- *pa.scheduler.policy*: Scheduler default policy full name (default is org.ow2.proactive.scheduler.policy.PriorityPolicy). The policy specifies here has to be in the Scheduler classpath.
- *pa.scheduler.forkedtask.security.policy*: Forked java task default security policy path. Uses to define the policy of the forked task (default is config/scheduler/forkedJavaTask/forkedTask.java.policy).
- *pa.scheduler.core.jmx.connectorname*: Name of the JMX Connector for the Scheduler (default is 'JMXSchedulerAgent')
- *pa.scheduler.core.jmx.port*: Port number used by JMX. This port is used only for JMX service and the RMI protocol. It will create a RMI registry if needed.
- *pa.scheduler.account.refreshrate*: Accounting refresh rate from the database. Amount of time between request to the DB for accounting purpose. (Value specified in seconds, default is 10)
- *pa.scheduler.core.usersessiontime*: user is automatically disconnect after this time if no request is made to the scheduler, it can be used to restrict the duration of a session. Negative number indicates that session is infinite (value specified in second)
- *pa.scheduler.core.starttask.timeout*: Timeout for the start task action. Time during which the scheduling process could be waiting. This value relies on the system and network capacity (Value specified in milliseconds, default is 2000)
- *pa.scheduler.core.starttask.threadnumber*: Maximum number of threads used for the start task action. This property define the number of blocking resources until the scheduling loop will block as well. (default is 5)
- *pa.scheduler.core.listener.threadnumber*: Maximum number of threads used to send events to clients. This property defines the number of clients than can block at the same time. If this number is reached, every clients won't receive events until a thread unlock. (default is 5)

Jobs properties

- *pa.scheduler.job.factor*: Number used to create task IDs. If the job number is 123 and it contains 456 tasks and this property is set to 1000, then the task ID will be 123456. If this property is set to 10000, the task ID will be 1230456. Task ID is (jobId*this_factor +taskId). (default is 10000)
- *pa.scheduler.core.removejobdelay*: time interval between the retrieval of a job result and its suppression from the Scheduler specified in second. Set this time to 0 if you don't want the job to be remove anyway. (default is 3600)
- *pa.scheduler.core.automaticremovejobdelay*: Automatic remove job delay. (The time between the termination of the job and removing it from the scheduler). Set this time to 0 if you don't want the job to be remove automatically. (Value specified in seconds, default is 0)
- *pa.scheduler.job.removeFromDataBase*: According to the previous property, removes the job also in dataBase when removing it from scheduler. (default is false)

Tasks properties

- *pa.scheduler.task.initialwaitingtime*: time to wait for when a task has had a faulty state, specified in millisecond. For performance reason, if the task is faulty, it is not restarted immediately. This property defines the initial time to wait. Next waiting time is computed following this function: newTimeToWait=previousTimeToWait+n*1000 where n is the number of re-execution of the task. The new time to wait is capped to 60000ms. (default is 1000ms)
- *pa.scheduler.task.numberofexecutiononfailure*: number of execution allowed if a task failed. The difference between faulty and failed is the state of the resource on which the task is executed. Most of the time, a failure is detected if a task kill a resource. In this case, this property defines the number of execution allowed for this kind of potentially harmful task. (default is 2, so one retry)

DataSpaces properties

- *pa.scheduler.dataspace.defaultinputurl*: Default INPUT space URL. Used to define INPUT space of each job that does not define an INPUT space. This URL can be HTTP, FTP, file system provided by ProActive: PAPRMI...
- *pa.scheduler.dataspace.defaultinputurl.localpath*: Default INPUT space path. Used to define the same INPUT space but with a local (faster) access (if possible).
- *pa.scheduler.dataspace.defaultinputurl.hostname*: Host name from which the input localpath is accessible.
- *pa.scheduler.dataspace.defaultoutputurl*: Default OUTPUT space URL. Used to define OUTPUT space of each job that does not define an OUTPUT space. This URL can be, FTP, file system provided by ProActive: PAPRMI... and cannot be HTTP
- *pa.scheduler.dataspace.defaultoutputurl.localpath*: Default OUTPUT space path. Used to define the same OUTPUT space but with a local (faster) access (if possible).
- *pa.scheduler.dataspace.defaultoutputurl.hostname*: Host name from which the output localpath is accessible.

Logs properties

- *pa.scheduler.logs.provider*: full class name of logs forwarding method. Logs forwarding is the system used to forward logs from task execution to user. Possible values are:
  - org.ow2.proactive.scheduler.common.util.logforwarder.providers.SocketBasedForwardingProvider for simple socket. This one uses simple Java socket.
  - org.ow2.proactive.scheduler.common.util.logforwarder.providers.SocketWithSSHTunnelBasedForwardingProvider for SSH tunneled socket. This one uses Java socket with SSH tunneling for secure communication.
  - org.ow2.proactive.scheduler.common.util.logforwarder.providers.ProActiveBasedForwardingProvider for ProActive based communication. This one uses ProActive communication, safer but slower.

  (Default is org.ow2.proactive.scheduler.common.util.logforwarder.providers.ProActiveBasedForwardingProvider)

Authentication properties

- *pa.scheduler.auth.jaas.path*: Jaas module configuration, describing which authentication method are available. (default is config/authentication/jaas.config)
- *pa.scheduler.auth.privkey.path*: Scheduler private key, to decrypt credentials encrypted on client side with the related public key. Once decrypted, credentials are passed to the appropriate Jaas module. (default is config/authentication/keys/priv.key)
- *pa.scheduler.auth.pubkey.path*: Scheduler public key, used to encrypt clear credentials on user side before making them transit on the network. Needs to be known on server side so that the Scheduler can offer the key to clients asking for it. (default is config/authentication/keys/pub.key);
- *pa.ldap.config.path*: LDAP Authentication configuration file path, used to set LDAP configuration properties. (default is config/authentication/ldap.cfg)
- *pa.scheduler.core.defaultloginfilename*: Login file name for file authentication method. (default is config/authentication/login.cfg)
- *pa.scheduler.core.defaultgroupfilename*: Group file name for file authentication method. (default is config/authentication/group.cfg)
- *pa.scheduler.core.authentication.loginMethod*: Property that defines the method that has to be used for logging users to the Scheduler. It can be one of the following values:
  - SchedulerFileLoginMethod to use file login and group management
  - SchedulerLDAPLoginMethod to use LDAP login management

  (default is SchedulerFileLoginMethod)

Resources manager related properties

- *pa.scheduler.resourcemanager.authentication.credentials*: Path to the Scheduler credentials file for RM authentication. (default is config/authentication/scheduler.cred)
- *pa.scheduler.resourcemanager.authentication.single*: Use single or multiple connection to RM : If true the scheduler user will do the requests to RM, if false each Scheduler users have their own connection to RM using their scheduling credentials (default is true)

Hibernate properties

- *pa.scheduler.db.hibernate.configuration*: Hibernate main configuration file (default is config/scheduler/database/hibernate/hibernate.cfg.xml)
- *pa.scheduler.db.hibernate.dropdb*: Drop database before creating a new one : default false If this value is true, the database will be dropped and then re-created If this value is false, database will be updated from the existing one.

3.2.2. Start the Scheduler using shell command

To start a local scheduler, run the **scheduler-start[.bat]** script in 'bin/[os]/' directory. Without arguments, the ProActive Scheduler will start on the local host and will try to connect to a started local Resources Manager with the default database configuration file. If no Resources Manager exists, it will create its own Resources Manager. Note that the database will be created if it does not exist.

scheduler-start[.bat] can be started with 2 optional arguments (use -h option to see a description of the available options):

- **The URL of a Resources Manager** already started (using the "-u URL" option). If you don't use this, it will try to connect to a started Resource Manager on the local host.

- **The scheduling policy** that will be started with the Scheduler (using "-p org.ow2.proactive.scheduler.policy.PriorityPolicy" option). By default, it will use the **PriorityPolicy** provided with the package.

For example, the following line will launch a scheduler on a Resource Manager started on "localhost" (which is in fact, the same as 'no argument'). It will also use the 'PriorityPolicy' scheduling policy:

```
scheduler-start[.bat] -u rmi://localhost/ -p org.ow2.proactive.scheduler.policy.PriorityPolicy
```

By default, the "config/authentication/" directory has to contain the two authentication files (group and login). The "config/scheduler/database/hibernate" directory has to contain the "hibernate.cfg.xml" configuration file.

It is also possible to launch the ProActive Scheduler with only one of these options or without option at all. For more informations, use -h (or --help) option (i.e. "scheduler-start[.bat] -h").

3.2.3. Start the Scheduler using the Java API

You can start the ProActive Scheduler using the Java API. Supposing that a Resource Manager is already started on toto (see Resource Manager documentation to create a resource Manager), here's a complete example showing how to start both Scheduler and Resource Manager (assuming the JVM is totally clean):

```
RMInitializer init = new RMInitializer();
init.setRMHomePath("/path/to/RM/home");
init.setLog4jConfiguration("/path/to/log4j");
init.setJavaSecurityPolicy("/path/to/securityPolicy");
init.setProActiveConfiguration("/path/to/PAconfigurationFile");
init.setResourceManagerPropertiesConfiguration("/path/to/RMPropertiesFile");
System.out.println("Starting RM, please wait...");
RMAuthentication rmAuth = RMFactory.startLocal(init);
System.out.println("RM successfully started at : " + rmAuth.getHostURL());

//starting scheduler...
//initializer is optionnal if the JVM already specified the following variables
SchedulerInitializer init = new SchedulerInitializer();
init.setSchedulerHomePath("/path/to/Scheduler/home");
init.setProActiveConfiguration("/path/to/PAconfigurationFile");
init.setSchedulerPropertiesConfiguration("/path/to/SchedulerPropertiesFile");
init.setPolicyFullClassName(PriorityPolicy.class.getName());

System.out.println("Starting Scheduler, please wait...");
SchedulerAuthenticationInterface sAuth = SchedulerFactory.startLocal(rmAuth.getHostURL(),init);
System.out.println("Scheduler successfully started at : " + sAuth.getHostURL());
```

Every thing could be started in different JVM and also without using the initializer. The example below shows how to start both RM and Scheduler assuming that the following variables are defined at JVM start up :

- **-Djava.security.manager :** to activate the security manager
- **-Dproactive.configuration=/path/to/pa_conf.xml :** to set the location of the PA Configuration file
- **-Djava.security.policy=/path/to/security_file :** to set the path of the java security policy file
- **-Dlog4j.configuration=file:/path/to/log4j_file :** to set the file of the log4j configuration file (ex: scheduler-log4j-server). The **file:** protocol is mandatory for this property.
- **-Dpa.rm.home=/path/to/rm_home :** the root path of the RM install.
- **-Dpa.scheduler.home=/path/to/scheduler_home :** the root path of the Scheduler install.

```
System.out.println("Starting RM, please wait...");
RMAuthentication rmAuth = RMFactory.startLocal();
System.out.println("RM successfully started at : " + rmAuth.getHostURL());

//starting scheduler...
System.out.println("Starting Scheduler, please wait...");
SchedulerAuthenticationInterface sAuth =
 SchedulerFactory.startLocal(rmAuth.getHostURL(),PriorityPolicy.class.getName());
System.out.println("Scheduler successfully started at : " + sAuth.getHostURL());
```

NOTE: By default, authentication are provided by the 2 files into the **config/authentication** directory. This files contains username, password and their groups. A LDAP authentication module is also available to replace authentication files (login.cfg and group.cfg) security module by a LDAP security module. To do so, just go into the **config/scheduler/settings.ini** configuration file and change the value of the property **pa.scheduler.core.authentication.loginMethod** to "SchedulerLDAPLoginMethod". Information about your own LDAP configuration can be set in the **ldap.cfg** into the "config/authentication" directory.

Another way is to start the ProActive Scheduler AND connect an administrator at the same time:

```
SchedulerFactory.createScheduler(
   credentials,
   new URI("rmi://host:port/"),
   "org.ow2.proactive.scheduler.policy.PriorityPolicy");
```

where arguments are the sames plus a **credentials** that authenticate the administrator who wants to connect. The user has to be in the login file and his group has to be 'admin' in the group file.

3.3. About job submission

According to the user manual, once connected, a user is ready to submit jobs. Here is a short explanation that describes the mechanism of submission. The **Authentication interface entity is no longer used** for this connected user. Figure 3.3, “A job submission” shows what happens when the Scheduler received a new job to schedule.

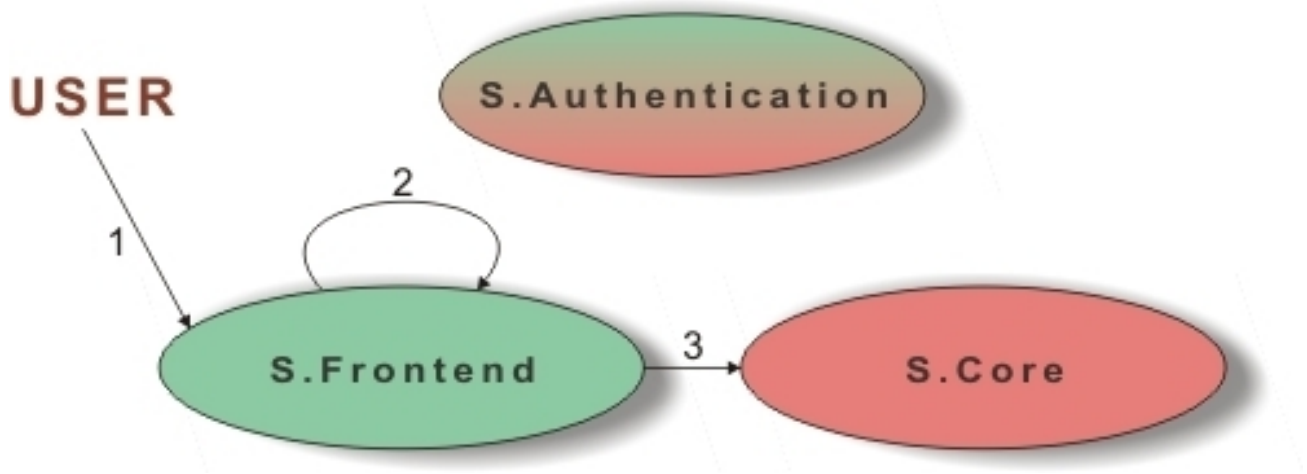


Figure 3.3. A job submission

1. First, the user submits a Job using his **UserSchedulerInterface** retrieved by the **logAsUser** method. Let's remind it (see Section 2.6.3, “Submit a job using the Java API” for how to use Credentials):

```
// connecting to the scheduler
SchedulerAuthenticationInterface auth = SchedulerConnection.join("rmi://localhost");
// checking username and password
Scheduler scheduler = auth.login(Credentials.getCredentials());
// submitting a new job and get the associated id
JobId myJobId = scheduler.submit(job);
```

2. The Scheduler Front-end checks the integrity of the job, and builds it in order to be ready to be managed by the Scheduler Core. If there is a problem, an exception is thrown explaining the cause of this problem.
3. Finally, the job is transmitted to the Core for scheduling.

3.4. Administer the ProActive Scheduler

As an administrator, it is possible to ask the Scheduler to start and stop, to kill jobs, set jobs priority, and so on... To start a command line administrator, go on to the next section. To use the Java API to manage it, please refer to Section 3.4.2, “Administer the Scheduler using the Java API”.

3.4.1. Administer the Scheduler using shell command

The admin Scheduler controller is an interface which allows to administer the ProActive Scheduler without the java API. It is also possible to see exceptions coming from the scheduler to know what happened. To start a admin controller, run the **scheduler-client[.bat]** script in 'bin/[os]/' directory. It works in the same manner as the user controller (Section 2.9, “Using the Scheduler controller”) but some administration functions have been added. Note that the admin controller uses administrator rights to allow the connection to the scheduler.

This controller provides additional controls compared to the user one. Here is the help of the admin one. To understand how to use it, please refers to the Section 2.9, “Using the Scheduler controller” section.

Here is the displayed help when using **scheduler-client[.bat] -h** command:

```
usage: scheduler-client [-c <arg>] [-cmd] [-cmdf] [-env <filePath> | -freeze | -jo <jobId> | -jp <jobId newPriority> | -jr <jobId> |
       -js <jobId> | -kill | -kj <jobId> | -lj | -lrm <rmURL> | -ma | -p <fullName> | -pause | -pj <jobId> | -resume | -rj <jobId> | -rmj
       <jobId> | -rp | -s <XMLDescriptor> | -sf <filePath arg1=val1 arg2=val2 ...> | -shutdown | -start | -stats | -stop | -test | -to <jobId
       taskName> | -tr <jobId taskName> | -ua <username>]  [-h] [-jn <jobName>]     [-k <sshkeyFilePath>]   [-l <login>]    [-o <logFile>]
       [-ss <selectScript>]       [-u <schedulerURL>]
 -c,--credentials <arg>                              Path to the credentials (/home/jlscheef/.proactive/security/creds.enc).
 -cmd,--command                                      <ctl> If mentionned, -submit argument becomes a command line, ie: -submit command
                                                     args...
 -cmdf,--commandf                                    <ctl> If mentionned, -submit argument becomes a text file path containing command
                                                     lines to schedule
 -env,--environment <filePath>                       Execute the given script as an environment for the interactive mode
 -freeze,--schedulerfreeze                           <ctl> Freeze the Scheduler (cause all non-running tasks to be paused)
 -h,--help                                           Display this help
 -jn,--jobname <jobName>                             <ctl> Used with -cmd or -cmdf, specify the job name
 -jo,--joboutput <jobId>                             <ctl> Get the output of the given job
 -jp,--jobpriority <jobId newPriority>               <ctl> Change the priority of the given job (Idle, Lowest, Low, Normal, High,
                                                     Highest)
 -jr,--jobresult <jobId>                             <ctl> Get the result of the given job
 -js,--jobstate <jobId>                              <ctl> Get the current state of the given job (Also tasks description)
 -k,--key <sshkeyFilePath>                           (Optional) The path to a private SSH key
 -kill,--schedulerkill                               <ctl> Kill the Scheduler
 -kj,--killjob <jobId>                               <ctl> Kill the given job (cause the job to finish)
 -l,--login <login>                                  The username to join the Scheduler
 -lj,--listjobs                                      <ctl> Display the list of jobs managed by the scheduler
 -lrm,--linkrm <rmURL>                               <ctl> Reconnect a RM to the scheduler
 -ma,--myaccount                                     <ctl> Display current user account information
 -o,--output <logFile>                               <ctl> Used with submit action, specify a log file path to store job output
 -p,--policy <fullName>                              <ctl> Change the current scheduling policy
 -pause,--schedulerpause                             <ctl> Pause the Scheduler (cause all non-running jobs to be paused)
 -pj,--pausejob <jobId>                              <ctl> Pause the given job (pause every non-running tasks)
 -resume,--schedulerresume                           <ctl> Resume the Scheduler
 -rj,--resumejob <jobId>                             <ctl> Resume the given job (restart every paused tasks)
 -rmj,--removejob <jobId>                            <ctl> Remove the given job
 -rp,--reloadpermissions                             <ctl> Reloads the permission file
 -s,--submit <XMLDescriptor>                         <ctl> Submit the given job XML file
 -sf,--script <filePath arg1=val1 arg2=val2 ...>     <ctl> Execute the given javascript file with optional arguments.
 -shutdown,--schedulershutdown                       <ctl> Shutdown the Scheduler
 -ss,--selectscript <selectScript>                   <ctl> Used with -cmd or -cmdf, specify a selection script
 -start,--schedulerstart                             <ctl> Start the Scheduler
 -stats,--statistics                                 <ctl> Display some statistics about the Scheduler
 -stop,--schedulerstop                               <ctl> Stop the Scheduler
 -test                                               <ctl> Test if the Scheduler is successfully started by committing some examples
 -to,--taskoutput <jobId taskName>                   <ctl> Get the output of the given task
 -tr,--taskresult <jobId taskName>                   <ctl> Get the result of the given task
 -u,--url <schedulerURL>                             The scheduler URL (default rmi://localhost/)
 -ua,--useraccount <username>                        <ctl> Display account information by username

NOTE : if no <ctl> command is specified, the controller will start in interactive mode.
```

Figure 3.4. Scheduler admin controller help

As shown, their are additional controls as start(), stop(), kill(), shutdown(), etc... An administrator can also interact on every job while the user can only manage jobs that belong to him.

3.4.2. Administer the Scheduler using the Java API

Let's first explain the connection mechanism. A user can connect to the ProActive Scheduler only if he/she is known. That is the goal of the authentication interface which is able to authenticate users. Following figure shows how the ProActive Scheduler connects a user.

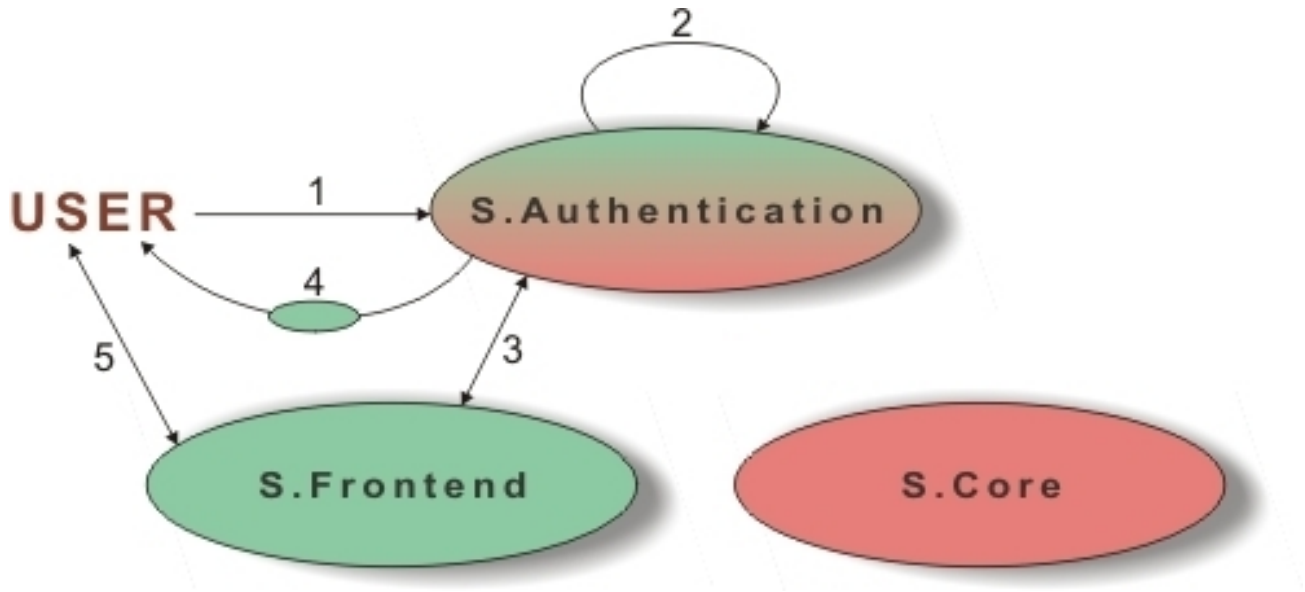


Figure 3.5. A user connection

1. First of all, a user tries to join the authentication interface using the **SchedulerConnection.join(...)** static method. It is also possible to use the SchedulerConnection.waitAndJoin(...) that will join the scheduler and wait for the connection to be established or an error to be raised.

   ```
   SchedulerAuthenticationInterface auth = SchedulerConnection.join("//host/");
   ```

   Then, with the **SchedulerAuthenticationInterface**, users can be connected as client using the **login** method (see Section 2.6.3, "Submit a job using the Java API" for how to use Credentials).

   ```
   Scheduler scheduler = auth.login(Credentials.getCredentials());
   ```

2. Next, the authentication object checks the users rights and whether the user is authorized to connect the scheduler. If not, an exception will be thrown.
3. Once connected, the authentication object sends the right and username/password to the scheduler front-end which will be able to authenticate user on its own.
4. If nothing goes wrong, the authentication interface will return a **Scheduler** which is in fact a direct link to the **Front-end**.
5. The client is now able to interact with the Scheduler using the returned interface.

And now, to administer the ProActive Scheduler, just connect to it if it is not already done and use the **Scheduler** to manage it. Take a look at the JavaDocumentation to get more details and features. Let's remind how to connect the Scheduler as an administrator (see Section 2.6.3, "Submit a job using the Java API" for how to use Credentials):

```
SchedulerAuthenticationInterface auth = SchedulerConnection.join("//host/");
Scheduler scheduler = auth.login(Credentials.getCredentials());
```

Then, use the returned 'scheduler' object to communicate with the ProActive Scheduler as an administrator. One interesting thing is to change the policy during the scheduling, that will consequently change the remaining scheduling order. Refer to the Section 3.9.1, "Add a new scheduling policy" to make your own policy and change it as shown below:

```
//scheduler is the AdminSchedulerInterface returned in the previous sections
scheduler.changePolicy(org.ow2.proactive.scheduler.policy.PriorityPolicy.class);
```

3.5. Configuring DataSpaces

The only things to configure in the DataSpace are the default INPUT and OUTPUT spaces. If not configured, it will use the default temporary directory of the system. To the default tmp dir, "scheduling/" will first be appended and then the 'defaultinput' and 'defaultoutput' directories. Finally, the username of the job owner will be also added to the previous path. For instance, if "defaultinput" is set to "myInput" and my username is "myUserName", then my INPUT data space will be **/tmp/scheduling/myInput/myUserName** (if the temporary directory is /tmp).

For example, to set the default INPUT space of the scheduler, just set the 3 following properties:

- **pa.scheduler.dataspace.defaultinputurl=http://myFileServer.myCompagny.com/files**
- **#pa.scheduler.dataspace.defaultinputurl.localpath** leave it commented for this example.
- **#pa.scheduler.dataspace.defaultinputurl.hostname** leave it commented for this example.

It is also possible to set a remote files system as a server of files thanks to the ProActive data server. In this new example, just locate the path to be served and start the ProActive data server: **pa-dataserver[.bat] D:\scheduling\outputSpace myOutput**. The first argument is the absolute path to the root space directory, the second one is the name of the server (could be useful to start more than one server on the same host).

Once launched, the server will retrieve an URL of the form: **paprmi://toto.company.com:1099/myOutput?proactive_vfs_provider_path=/**. Just put this URL in the scheduler settings:

- **pa.scheduler.dataspace.defaultoutputurl=paprmi://toto.company.com:1099/myOutput?proactive_vfs_provider_path=/**
- **pa.scheduler.dataspace.defaultoutputurl.localpath=D:\scheduling\outputSpace**
- **pa.scheduler.dataspace.defaultoutputurl.hostname=toto** this value allows a task executed on 'toto' to handle files using the direct access (localpath above). This could be great for performance issues.

3.6. Configuring task for execution under user account

To allow scheduler users to run task under their account login and password, administrator must configure workers machines (user name//password must be the same as system user name//password) :

1. **At system level :** Every host machine where tasks will be executed under user account must accept such an execution, the following describe the different system requirement :
   - **On unix system :** There are 3 possible ways to allow a user to do so. For each method, user account must be accessible (created or mount) from host (node) :
     a. *Use login authentication only :* User login (that is scheduler login also) must be put in sudoer list. In this case, scheduler user could start process using his login only.
     b. *Use login//password authentication :* User login and password (that is scheduler login//password also) will be used to authenticate user on the node. Requirement is just a valid unix account.
     c. *Use SSH key authentication :* User login and password are not used here, administrator must generate ssh key pair, add public key in authorized key and give private key to users. SSH localhost will be used for this method. User must provide (to the scheduler) a credentials containing the corresponding private ssh key. Here's an example of a command to create such a credential :

        bin/unix/create-cred -F config/authentication/keys/pub.key -l username -p userpwd -k path/to/private/sshkey -o myCredentials.cred : This command will create a new credentials with "username" as login, "userpwd" as password, using scheduler public key at "config/authentication/keys/pub.key" for credentials encryption and using the private ssh key at "path/to/private/sshkey" provided by administrator. The new credential will be stored in "myCredentials.cred"

- **On Windows system :** User account must be created on the windows host with same username//password as the corresponding scheduler user account. The process running the node must be started under administrator user with 2 needed privileges (default windows configuration) : SE_ASSIGNPRIMARYTOKEN_NAME and SE_INCREASE_QUOTA_NAME.

2. **At node level :** The node must be configured by setting the method of system authentication the administrator wants to use. It is done by setting a Java property at node startup, *pa.laucher.forkas.method* which value can be :
   - "none" : no method is used, user cannot execute task under his account
   - "pwd" : password method is used, user will start the task using his scheduling login and password. On Unix, if sudoer list is used, user will be authenticated using his username only.
   - "key" : ssh private key is used, user must provide a credentials containing login//password and a private ssh key provided by administrator to allow user to execute process with ssh on localhost.

3.7. Accounting

The Scheduler keeps track on how much jobs and tasks were done by a user. All the values are computed since the creation of the database.

- The total job count: the count of all jobs initiated by a user.
- The total job duration: the overall amount of time spent in job execution initiated by a user.
- The total task count: the count of all tasks initiated by a user.
- The total task duration: the overall amount of time spent in task execution.

The accounting information can be accessed through a JMX client or through the Scheduler GUI.

3.8. Connecting to JMX as administrator

The JMX interface for remote management and monitoring provides information about the running ProActive Scheduler and allows the user to modify its configuration. For more details about JMX concepts please refer to official documentation about the JMX architecture[1].

[1] http://java.sun.com/j2se/1.5.0/docs/guide/jmx/overview/architecture.html

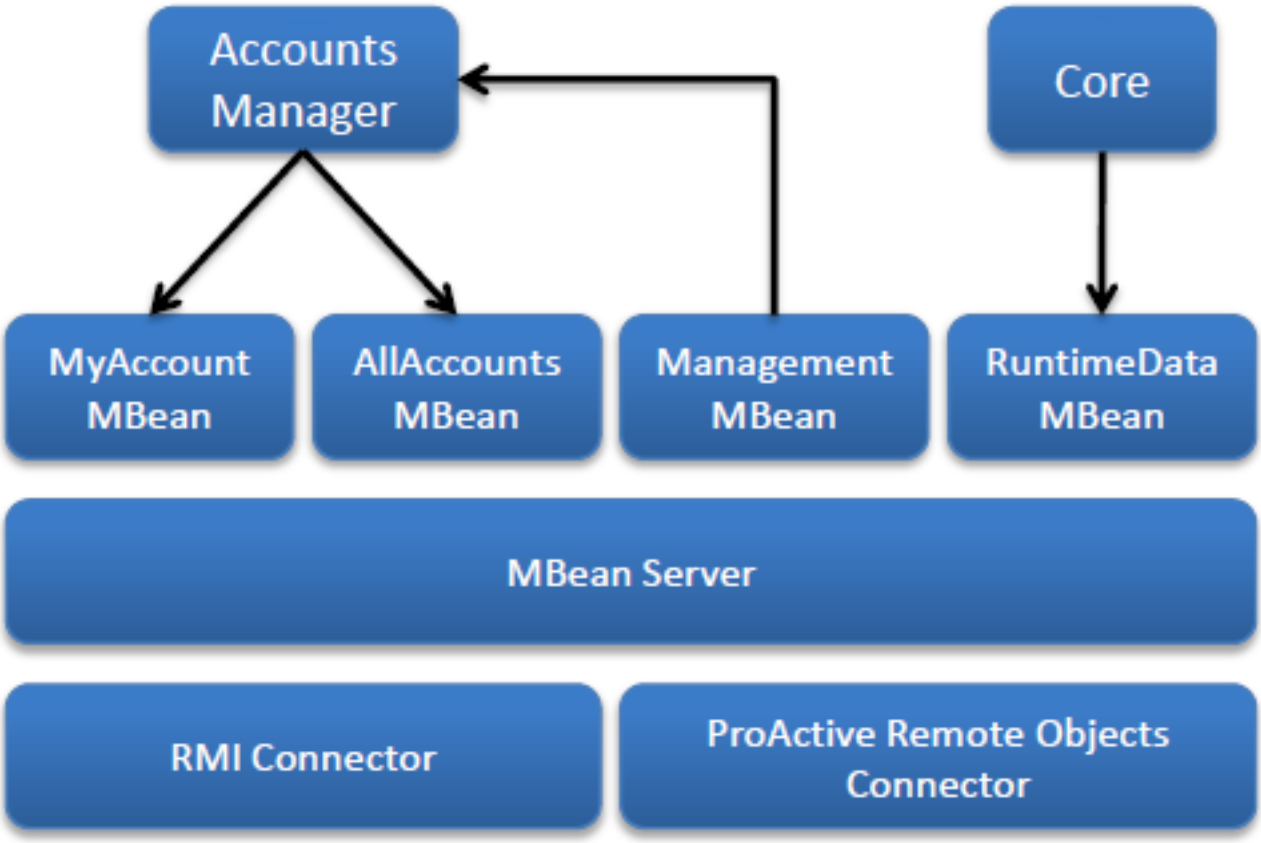


Figure 3.6. Structure of the Scheduler JMX interface

The following aspects (or services) of the ProActive Scheduler are instrumented using MBeans that are managed through a JMX agent.

- The Scheduler **Core** exposes various runtime information using the **RuntimeDataMBean** such as:
  - The Scheduler status
  - Pending/Running/Finished/Total job and task count
  - Some average values
- The **Accounts Manager** exposes accounting information for a specific user with the **MyAccountMBean** and **AllAccountsMBean** such as:
  - The total job and task count
  - The total job and task duration
- Various management operations are exposed using the **ManagementMBean** such as:
  - Setting the accounts refresh rate
  - Refresh all accounts
  - Reload the permission policy file

As shown on Figure 3.6, “Structure of the Scheduler JMX interface” the MBean server can be accessed by remote applications using one of the two available connectors:

- The standard solution based on Remote Method Invocation (RMI) protocol is the RMI Connector accessible at the following url:

  service:jmx:rmi:///jndi/rmi://**HOSTNAME**:**PORT**/JMXSchedulerAgent

  where

- **HOSTNAME** is the hostname on which the Scheduler is started
- **PORT** (5822 by default) is the port number on which the JMX RMI connector server has been started, it is defined by the property **pa.scheduler.jmx.port**

- The ProActive Remote Objects Connector provides ProActive protocol aware connector accessible at the following url:

  service:jmx:ro:///jndi/**PA_PROTOCOL**://**HOSTNAME**:**PORT**/JMXSchedulerAgent

  where

  - **PA_PROTOCOL** is the protocol defined by the **proactive.communication.protocol** property
  - **HOSTNAME** is the hostname on which the Scheduler is started
  - **PORT** is the protocol dependent port number usually defined by the property **proactive.PA_PROTOCOL.port**

The name of the connector (JMXSchedulerAgent by default) is defined by the property **pa.scheduler.jmx.connectorname**.

The JMX url on which to connect can obtained from the Authentication API of the Scheduler or by reading the log file located in $SCHEDULER_HOME/.logs/SchedulerDev.log. In the log file, the address you have to retrieve is the one where the JMX RMI connector server has been started. Once connected, you'll get an access to Scheduler statistics and accounting.

For example, to connect to the Scheduler JMX Agent with the popular JConsole tool, just enter the url of the standard RMI Connector as shown on the Figure 3.7, “Connection using JConsole”, as well as the username and the password.

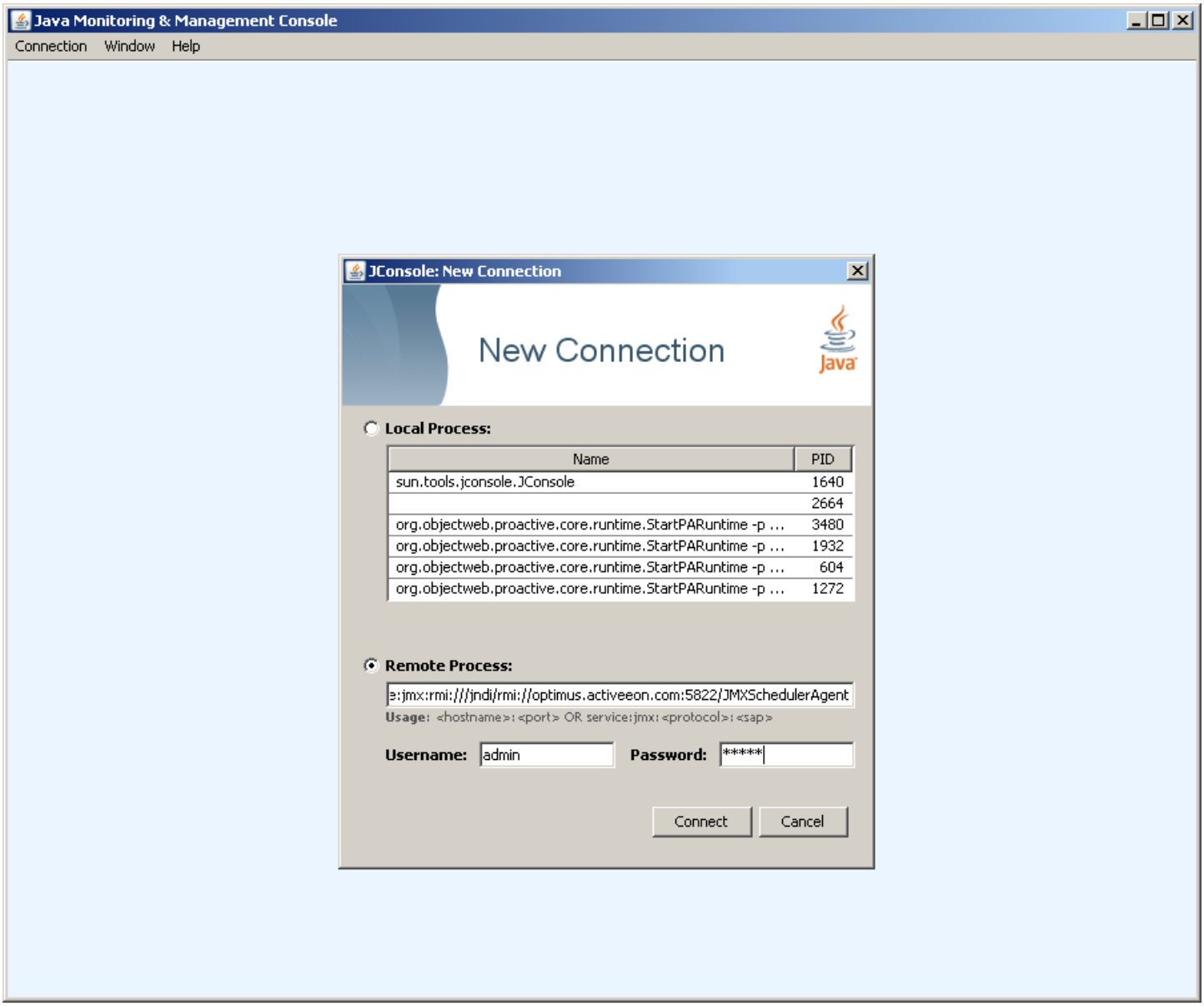


Figure 3.7. Connection using JConsole

Then depending on the allowed permissions browse the attributes of the MBeans.

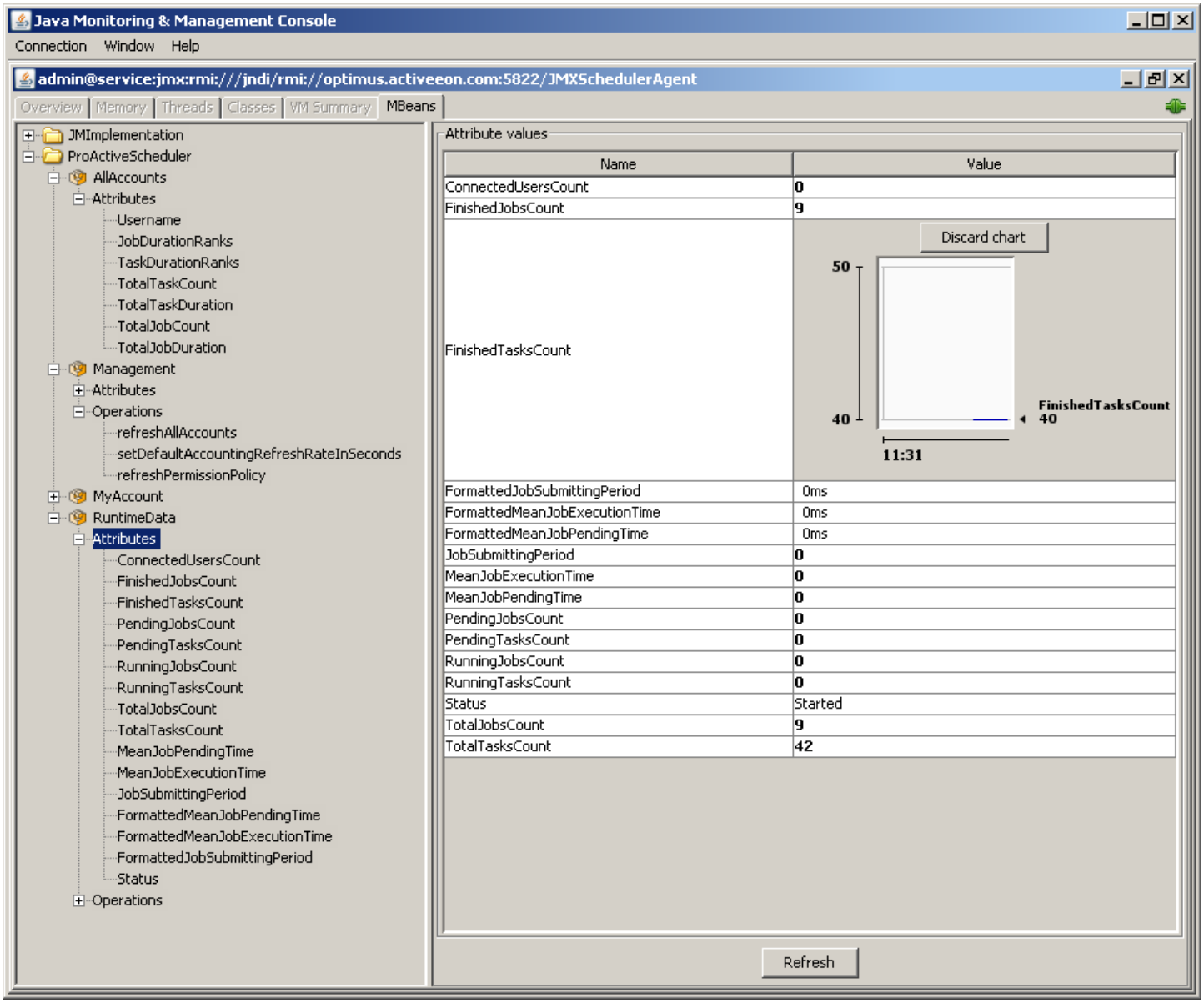


Figure 3.8. Browse MBean attributes

3.9. Extend the ProActive Scheduler

3.9.1. Add a new scheduling policy

Create and add a new scheduling policy remains a very simple work. You just have to implements the **org.ow2.proactive.scheduler.common.policy.Policy** interface and start a new Scheduler with this new policy as argument, or dynamically change it if the scheduler is already running.

Here is the interface which has to be implemented. The default implementation is **PriorityPolicy** in the same package:

```
public abstract class Policy implements Serializable {
  //Resource Manager state field.
```

```
    public RMState RMState = null;
    //method to implement
    public abstract Vector<EligibleTaskDescriptor> getOrderedTasks(List<JobDescriptor> jobs);
}
```

This method returns all the tasks that have to be scheduled. Tasks must be in the desired scheduling order. The first task to be scheduled will be the first in the returned Vector.

The parameters is a list of running and pending jobs, which contain tasks to be scheduled. Some properties in each job and task can be accessed in order to make your own scheduling order. It is also possible to access to the "RMState" field that allows you to have information about resources like used nodes, total nodes number, etc. The only thing to do is to **extract the task**, **re-order them** and **put them in a vector**. Let's see the default implementation of the ProActive Scheduler policy to illustrate it:

```
public class PriorityPolicy implements PolicyInterface {

    /**
     * This method returns the tasks using FIFO policy according to the jobs priorities.
     *
     * @see org.objectweb.proactive.extensions.scheduler.policy.PolicyInterface#getReadyTasks(java.util.List)
     */
    @Override
    public Vector<EligibleTaskDescriptor> getOrderedTasks( List<JobDescriptor> jobs) {

        Vector<EligibleTaskDescriptor> toReturn = new Vector<EligibleTaskDescriptor>();

        //sort jobs by priority
        Collections.sort(jobs);

        //add tasks to the list to return
        for (JobDescriptor lj : jobs) {
            toReturn.addAll(lj.getEligibleTasks());
        }

        //return
        return toReturn;

    }
}
```

By default, jobs know how to be sorted regarding to their priorities. But it is simpler to create a Comparator and sort jobs with it. It is also possible to have some information about resources using the protected **RMState** field inside the getOrderedTasks method.

3.10. Configure users authentication

As presented before, scheduler users are authenticated at scheduler connection. Users have to enter a login and a password checked by Scheduler. There are two authorization levels:

- **user level:** enables to submit jobs (see waiting queue, jobs running and jobs finished). They can change priority of their jobs, with 3 priority levels: lowest, low and normal. They are able to see their jobs output, cancel and retrieve results only from their own jobs. They cannot perform actions on jobs submitted by other users.
- **admin level:** enables to pause, stop and restart scheduler, cancel or retrieve results of any job, change priority of any job, with 5 priority levels: lowest, low, normal, high and highest.

3.10.1. KeyPair authentication

Regardless of which method is actually used to perform the authentication, credentials need to be passed from the client to the Scheduler, through the network. The data will be encrypted with an AES symmetric secret key to allow unlimited credentials size, and the AES key itself will be encrypted with an RSA keypair.

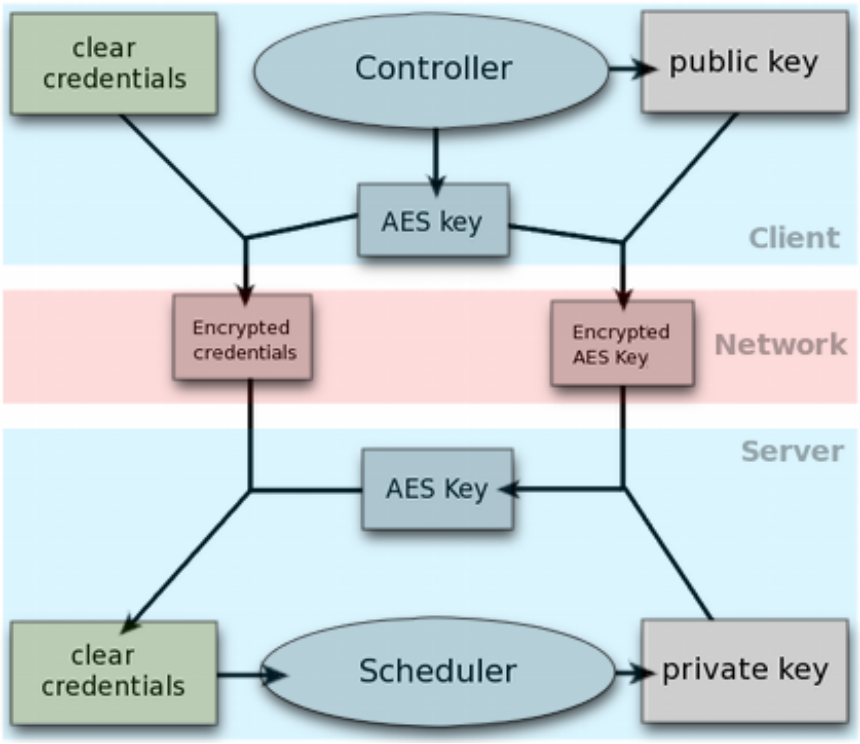


Figure 3.9. Credentials encryption

The Keypair can be generated with the **key-gen[.bat]** script:

```
bin/unix$ ./key-gen -p $HOME/.proactive/priv.key -P $HOME/.proactive/pub.key
```

Accordingly, the Scheduler configuration has to be set so that, when started:

- **pa.scheduler.auth.privkey.path**=$HOME/.proactive/priv.key
- **pa.scheduler.auth.pubkey.path**=$HOME/.proactive/pub.key

Although no encryption should be performed on server side, the public key should be known from the Scheduler: indeed, a client can request the public key to the Scheduler so that it may encrypt its credentials to perform authentication. This method does not require the administrator of the Scheduler to manually propagate public keys to all its users. Users can encrypt theirs credentials with the **create-cred[.bat]** script. See Section 2.6.3, “Submit a job using the Java API” for client-side configuration.

3.10.2. Select authentication method

So that the Scheduler can manage users authentication and authorization, it has to store users account/password, and check login and password at users' connection. This storage of user accounts can be managed in two ways: by files, or by LDAP. A Scheduler property (in config/scheculer/settings.ini) specifies which kind of authentication will be used:

```
#Property that defines the method that has to be used for logging users to the Scheduler
```

```
#It can be one of the following values:
#    - "SchedulerFileLoginMethod" to use file login and group management
#    - "SchedulerLDAPLoginMethod" to use LDAP login management
pa.scheduler.core.authentication.loginMethod=SchedulerFileLoginMethod
```

By default, authentication method is by file (SchedulerFileLoginMethod). If you want to use the LDAP-based authentication, replace the "SchedulerFileLoginMethod" value by "SchedulerLDAPLoginMethod".

3.10.3. Configure file-based authentication

By default, the Scheduler stores user accounts, passwords, and group memberships (user or admin), in two files:

- **config/authentication/login.cfg** stores the user names and passwords of accounts. Each line has to look like **user:passwd**. The default login.cfg file is given hereafter:

```
admin:admin
demo:demo
user:pwd
test:pwd
radmin:pwd
nsadmin:pwd
provider:pwd
scheduler:scheduler_pwd
```

- **config/authentication/group.cfg** stores the user names of memberships. For each user registered in login.cfg, a group membership has to be defined in this file. Each line has to look like **user:group**. Group has to be **user** to have user rights, or **admin** to have administrator rights. Default group.cfg file is like this:

```
admin:admin
demo:admin

provider:providers

user:user
scheduler:user

radmin:rmcoreadmins
nsadmin:nsadmins
admin:nsadmins
```

You can change the default paths of these two files. Edit the **config/scheduler/settings.ini** file and change the two following properties:

- **pa.scheduler.core.defaultloginfilename** - To define a user/password file, change this line as follows: pa.scheduler.core.defaultloginfilename=/path/to/the/file/mylogins.cfg
- **pa.scheduler.core.defaultgroupfilename** - To define a group membership file, change the line as follows: pa.scheduler.core.defaultgroupfilename=/path/to/the/file/mygroups.cfg

3.10.4. Configure LDAP-based authentication

Scheduler is able to connect to an existing LDAP to check users' login/password, and verify users' group membership. This authentication method can be useful if you have in your organization an LDAP that already stores user/password entries. There are several points to configure: path in LDAP where scheduler users and admins entries are stored, LDAP groups that define user and admin group membership, URL of the LDAP, LDAP binding method used by connection and configuration of SSL/TLS if you want a secured connection between Scheduler and LDAP. All LDAP connection parameters are set in **config/authentication/ldap.cfg**.

config/scheduler/settings.ini defines a path to a configuration file that contains all LDAP connection and authentication properties. Default value for this property defines a default configuration file: **config/authentication/ldap.cfg**. Specify your LDAP properties (explained below) in this file.

3.10.4.1. Set LDAP url

First, you have to define the URL of your organisation's LDAP. This address corresponds to the property: **pa.ldap.url**. You have to put a standard LDAP-like URL, for example **ldap://myLdap**. You can also set an URL with secure access: **ldaps://myLdap:636**. See Section 3.10.4.5, “Set SSL/TLS parameters” for SSL/TLS configuration.

3.10.4.2. Set users subtree and login attribute

You have to define where users' entries are stored in LDAP tree, i.e. the path where Scheduler will try to find users' and admins' entries. The users subtree corresponds to the property: **pa.ldap.userssubtree**. Put a DN of your LDAP like this: pa.ldap.userssubtree=ou=myUsers,o=myOrganisation,dc=myFirm,dc=com.

Then, you have to specify in users' entries, the attribute that corresponds to the user login. This is done by the property: **pa.ldap.user.login.attr**. Put an attribute name like this: pa.ldap.user.login.attr=loginName.

3.10.4.3. Set users and admin groups

After having found a user entry in LDAP, and checked the user's password, LDAP checks the user's group membership. Your organization's LDAP has to provide two entries of type **groupOfUniqueNames**, that present attributes of type **uniqueMember**. This type corresponds to users' DN that has access to the Scheduler (typical group definition in a LDAP). You have to set the two entries of this type wherein users' and admins' DN are put in 'uniqueMemeber' attribute. Admins group DN is set in scheduler by the property **pa.ldap.admins.group.dn**. User group DN is set by the property **pa.ldap.users.group.dn**.

3.10.4.4. Configure LDAP authentication parameters

By default, the Scheduler binds to LDAP in anonymous mode. You can change this authentication method by modifying the property **pa.ldap.authentication.method**. This property can have several values:

- **none** (default value) - the Scheduler performs connection to LDAP in anonymous mode.
- **simple** - the Scheduler performs connection to LDAP with a specified login/password (see below for user password setting).
- You can also specify a SASL mechanism for LDAPv3. There are many SASL mechanisms available: **cram-md5**, **digest-md5**, **kerberos4**... Just put **sasl** to this property to let scheduler JVM choose SASL authentication mechanism.

If you specify an authentication method different from 'none' (anonymous connection to LDAP), you have to specify a login/password for authentication. There are two properties in config/scheduler/settings.ini to set:

- **pa.ldap.bind.login** - set user name for authentication.
- **pa.ldap.bind.pwd** - set password for authentication.

3.10.4.5. Set SSL/TLS parameters

Scheduler is able to communicate with LDAP with a secured SSL/TLS layer. It can be useful if your network is not trusted, and critical information are transmitted between scheduler and LDAP, such as users' passwords. First, set the LDAP URL property **pa.ldap.url** to a URL of type ldaps://myLdap. Then, set **pa.ldap.authentication.method** to none so as to delegate authentication to SSL.

For using SSL properly, you have to specify your certificate and public keys for SSL handshake. Java stores certificates in a keyStore and public keys in a trustStore. In most of cases, you just have to define a trustStore with public key part of LDAP's certificate. Put certificate in a keyStore, and public keys in a trustStore with the keytool command (keytool command is distributed with standard java platforms):

keytool -import -alias myAlias -file myCertificate -keystore myKeyStore

myAlias is the alias name of your certificate, **myCertificate** is your private certificate file and **myKeyStore** is the new keyStore file produced in output. This command asks you to enter a password for your keyStore.

Put LDAP certificate's public key in a trustStore, with the keytool command:

```
keytool -import -alias myAlias -file myPublicKey -keystore myTrustStore
```

myAlias is the alias name of your certificate's public key, **myPublicKey** is your certificate's public key file and **myTrustore** is the new trustStore file produced in output. This command asks you to enter a password for your trustStore.

Finally, in **config/authentication/ldap.cfg**, set keyStore and trustStore created before to theirs respective passwords:

- Set **pa.ldap.keystore.path** to the path of your keyStore.
- Set **pa.ldap.keystore.passwd** to the password defined previously for keyStore.
- Set **pa.ldap.truststore.path** to the path of your trustStore.
- Set **pa.ldap.truststore.passwd** to the password defined previously for trustStore.

3.10.4.6. Use fall back to file authentication

You can use simultaneously file-based authentication and LDAP-based authentication. Indeed, If the LDAP-based authentication fails, Scheduler can check users' password and group membership in login and group files, as performed in the FileLogin method. It uses the **pa.scheduler.core.defaultloginfilename** and **pa.scheduler.core.defaultgroupfilename** files to authenticate user and check group membership. There are two rules:

- If LDAP group membership checking fails, fall back to group membership checking with group file. To activate this behavior set **pa.ldap.group.membership.fallback** to **true** in the LDAP configuration file.
- If a user is not found in LDAP, fall back to authentication and group membership checking with login and group files. To activate this behavior, set **pa.ldap.authentication.fallback** to **true**, in LDAP configuration file.

3.10.5. Configure node to allow task execution under user system account

The ProActive Scheduling is also able to start task under system user account ID if configured. The mechanism requires some system administration but remains quite simple. The goal is to configure system to be able to accept user account connection (password or ssh) and then configured the node to specify what kind of authentication method it will accept :

1. **System configuration :** The system must be configured to accept user system connection. So you must add account to user which must have the same username as the name in Scheduler account. You can authorized user accessing host throught password and/or using ssh connection. For exemple, if you don't want to use password authentication, you can generate public and private couple of key, add the public key to 'authorized_keys' file in every host that will authorized this key and give the private key to every user that should be able to access those hosts.

   Note : if you use only password authentication, username/password of Scheduler users must be the same as system username/password.
2. **User configuration :** The only thing user must do is to create credentials including the private ssh key you've generated for him. The section Section 2.6.3, “Submit a job using the Java API” describe how a user can create his credentials using private ssh key.

   If not using ssh authentication, user has nothing special to do except setting the runAsMe property to 'true' in his tasks.
3. **Node configuration :** The node must also be configured to accept this king of execution. Configure the **node** is just setting a java property at node deployment. See Resource Manager documentation to know how to specify java properties at node deployment. If not configured (default) every "runAsUser" executions will be rejected. If configured, the behavior will depend on the following java property :
   - **pas.launcher.forkas.method=none :** the node won't accept execution under user account.

     pas.launcher.forkas.method=pwd : the node will only accept execution under user account using password authentication. In such a case, the username/password of the Scheduler user must be the same as the system one.

     pas.launcher.forkas.method=key : the node will only accept execution under user account using ssh authentication method. Scheduler username must be the same as the system account one, password can be different as the system authentication will

be done with ssh key. This method only work on unix system and is safer for system account. (system password is never sent to user, security is just based on a line in 'authorized_keys' file.)

Chapter 4. Scheduler Tutorial

4.1. ProActive Scheduler Tutorial

4.1.1. introduction

This chapter is a guide on using the ProActive Scheduler. The Scheduler is a tool for deployment, administration and maintenance of a job queue over a Grid or a P2P infrastructure and over various platforms following one of many set of rules regarding the job management.

The chapter is structured in the following sections:

- Scheduler architecture Section 4.1.2, “Scheduler architecture”
- Task flow concept: Section 4.1.3, “Task-flow Concept”
- Schedule a native executable: Section 4.1.4, “Schedule a native task”
- Launch Scheduler and submit a job:Section 4.1.5, “Launch the scheduler, submit a job and retrieve the result”
- Node selection mechanism: Section 4.2, “Adding a selection script to the task”
- PreScript and PostScript features: Section 4.3, “PreScript and PostScript”
- How to generate dynamically a native command to execute, depending on the node architecture: Section 4.4, “Command generator script”
- Exported Environment variables introduced into the task's execution environment: Section 4.5, “Using exported environment variables”

Warning

Prerequisites and conventions:

- You must have an Eclipse 3.2 IDE installed, and the ProActive Project open in your Eclipse.
- The directory of ProActive in your system project is represented by the tag: [ProActive_dir].
- You need a working directory in your system, represented here by [working_dir].

4.1.2. Scheduler architecture

The scheduler is made of two main components: the Scheduler and the Resource Manager. Each of them has its own functionality:

- The scheduler is in charge of registering jobs submitted and put them in a queue according to a scheduling policy. Then, it has to ask for resources at the Resource Manager, and execute jobs on those retrieved resources.
- The Resource Manager (RM) handles a set of available resource available for scheduling jobs. It benefits from the Proactive Library, so it can handle resources from LAN, on cluster of workstations, on P2P desktop Grids, or on Internet Grids. Resource Manager provides the scheduler with resources, according to criteria (Operating System, dynamic libraries, Memory...). Resources, at ProActive point of view, are called nodes. Resource Manager therefore supplies ProActive nodes to the Scheduler.

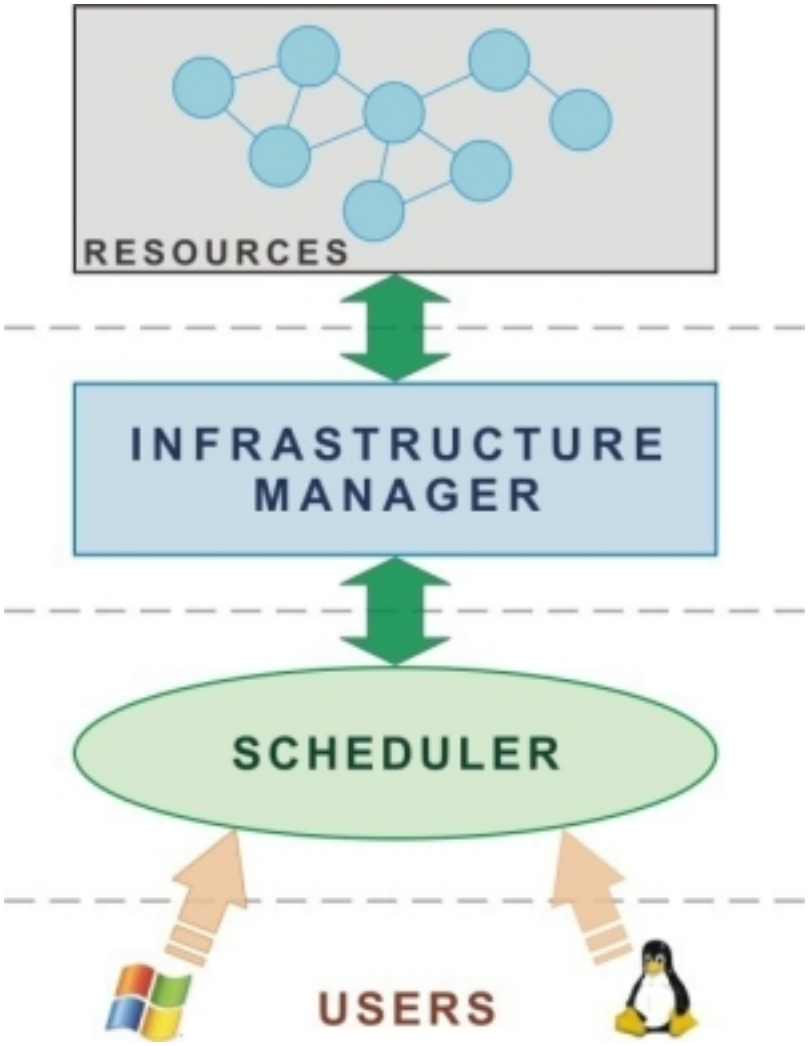


Figure 4.1. Scheduler architecture

These two components (Scheduler and RM) are independent and run as non GUI daemon/service. Thus, they can run on two different hosts.

4.1.3. Task-flow Concept

Submit a job to the ProActive Scheduler means submitting a **task flow**, which is a set of tasks. A task can be defined as a part of the job, i.e. a step to be executed in the job. Jobs can also be made of different tasks, i.e. different steps to execute. Task is the most little part of a job, the smallest schedulable entity. ProActive Scheduler has computing nodes to execute jobs and basically, it launches an execution of a task on each node. When task is ended, it launches another one and so on and so forth.

4.1.3.1. Parallel tasks, predecessor tasks

When you build your job, you create a graph of tasks, with a definition of predecessor/successor between tasks:

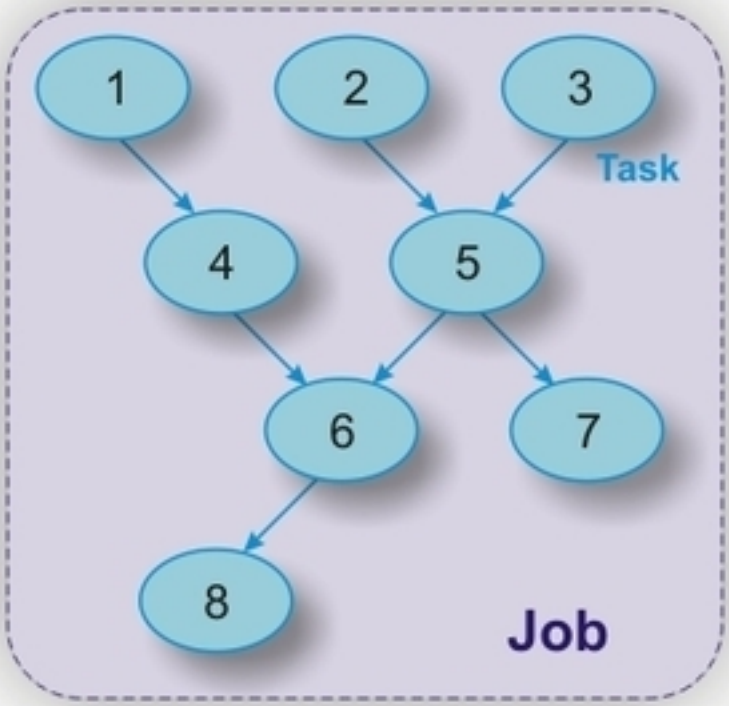


Figure 4.2. Scheduler architecture

In this tasks graph, we see that task 4 is preceded by task 1, that means scheduler waits the end of task 1 execution before launching task 4. In a more concrete way, task 1 could be the calculation of a part of the problem to solve, and task 4 takes result provided by task 1 and compute another step of the calculation. We introduce here the concept of **result passing** between tasks, explained later. This relation is called a dependence, and we say that task 4 **depends** on task 1.

We see that task 1, 2 and 3 are not linked, so these three tasks can be executed in **parallel**, because there are independent from each other.

Definition of the task-flow graph is made at job definition, before scheduling, and cannot be modified during the job execution, this kind of work flow is called a **static work flow**. A task flow job is described in XML language.

4.1.4. Schedule a native task

The scheduler provides the possibility to launch jobs containing tasks which are native executables, like a C/C++ program. The following example is a native C executable called nativTask which displays ten dots, waiting a number of seconds given in argument between each display.

4.1.4.1. Native C code of the task

Complete native C code of the task:

project: ProActive

directory: descriptors/deployments/scheduler/jobs/job_native_linux

file: nativTask.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

#define DEFAULT_WAIT_TIME 1

int main(int argc, char**argv){
	printf("Starting native task...\n");
	/* parsing args */
	int nb = DEFAULT_WAIT_TIME;
	if(argc > 1){
		printf("argv[1]= %s\n", argv[1]);
		sscanf(argv[1], "%d", &nb);
		printf("Waiting time : %d s\n", nb);
	}
	int i;
	for(i = 0; i < 10; i++){
		printf(".");
		fflush(stdout);
		sleep(nb);
	}
	printf("\nNative task terminated !\n");

	return EXIT_SUCCESS;
}
```

You have also an unix compiled version of this program named nativTask in the same directory. Compile the native code above (or take the already compiled version), and place the executable in your [working_dir]. Now, you need to write an XML file which describes the job.

4.1.4.2. XML job description

directory: [working_dir].

In this XML file, tasks composing the job and the "predecessor/successor" orders are specified. Create a file job_HelloWorld.xml in your [working_dir] and write the following code:

file: job_native.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
  name="job_nativ" priority="normal" logFile="${EXEC_PATH}/nativTask.log">
  <description>Will execute 2 native C tasks</description>
  <variables>
    <variable name="EXEC_PATH" value="[working_dir]"/>
  </variables>
  <taskFlow>
    <task name="task1" preciousResult="true">
      <description>Will display 10 dots every 1s</description>
      <nativeExecutable>
        <staticCommand
          value="${EXEC_PATH}/nativTask">
          <arguments>
```

```
                <argument value="1"/>
              </arguments>
            </staticCommand>
          </nativeExecutable>
        </task>
        <task name="task2" preciousResult="true">
          <description>Will display 10 dots every 2s</description>
          <depends>
              <task ref="task1"/>
          </depends>
          <nativeExecutable>
            <staticCommand
              value="${EXEC_PATH}/nativTask">
              <arguments>
                <argument value="2"/>
              </arguments>
            </staticCommand>
          </nativeExecutable>
        </task>
    </taskFlow>
</job>
```

Job tag:

Here are attribute functions:

- **schemaLocation** - XML schema related to a job descriptor.
- **name** - name of the job.
- **priority** - priority of the job.

  There are tree priority levels: lowest, low, normal. Actually, there are five levels, but the two other levels (high and highest) can only be setted by the administrator of the scheduler.
- **logFile** - path of the log file.

  Standard and error outputs of all launched native programs are written in this file. You can notice the use of the ${EXE_PATH} variable. That will be explained later.

Variables tag:

In this tag, the directory path variable called “EXEC_PATH” has been defined. Thus, ${EXEC_PATH} represents the string defined in this variable. This a convenient way for not repeating strings and for doing changes rapidly.

Description tag:

A literary human readable description of the job.

Taskflow tag:

Specifies that the job is made of tasks that need to be executed in a certain order.

Task tag:

Specifies a task of the job. The task name is used to identify the task from the others. This parameter is mandatory and has to be unique in the XML job descriptor. The parameter preciousResult="true" precises that the result is needed and have to be retrieved at the end of

the job. The task2 task contains a tag called **<depends>** which specifies a list of tasks that have to be executed and terminated before the execution of the task2. Therefore, task2 can be launched only if task1 is terminated.

Description tag of the task:

A literary description of the task.

nativeExecutable tag:

This tag is used in order to specify that the task is an execution of a native program. This tag has a sub-tag called staticCommand whose value represents the command to be executed. This sub-tag also contains a arguments tag where command parameters are written. Thus, you can specify parameters for execution as if you write parameters in a command line.

Now the job is defined and ready to be scheduled!

4.1.5. Launch the scheduler, submit a job and retrieve the result

In this part, you will learn how to launch the Scheduler program and submit your job made of the two native tasks. Two ways of launching the Scheduler will be exposed: a simple one which enables you to directly submit a job without worrying by the nodes handling and another one which is more flexible but a little bit longer. These actions are performed by scripts located into the directory: [Scheduler_Home_Dir]/bin/[OS]/

Note

We assume in the whole section that we are working on a Unix system. However, scripts for windows with the .bat extension are also available.

4.1.5.1. Launch the scheduler (easy way)

This way of doing enables you to launch a scheduler without manually launching a Resources Manager before and adding nodes to it. It uses either the **scheduler-start** script or the **scheduler-start-clean** one. Type the following command:

```
$ scheduler-start[-clean] -u //localhost/
```

The **"-u URL"** option specifies the location of a running Resource Manager. If you don't specify a URL for the RM, the Scheduler will first try to connect to a local Resource Manager (at the address //localhost/) and if it cannot find one, it will create a new one. You can choose either to use the previous database if you have already launched a scheduler before and to create a new one using the scheduler-start-clean script. This script will launch the Scheduler service, and the Scheduler will connect to the Resource Manager which has been previously launched. Once started, you can see on the standard output the line:

```
Scheduler successfully created on rmi://hostname:port/
```

4.1.5.2. Launch the Resource Manager and then the Scheduler (flexible way)

Launching the Resource Manager before the Scheduler gives you more flexibility. To do this, you have first to use the **rm-start** script. Type:

```
$ rm-start
```

That will start a Resource Manager with no node. Once started, you can see on the standard output the line:

```
Resource Manager successfully created on rmi://hostname:port/
```

Then, you can add as many nodes as you want. For this, you have to use both the **rm-start-node** and the **rm-client** scripts. So, you first have to start a new node. Type the following command to start a node called 'MyNode':

```
rm-start-node MyNode
```

You can see that your node has well been created and started when the line hereafter has been displayed:

```
OK. Node MyNode ( rmi://hostname:port/MyNode ) is created in VM id=5530b46abee98f42:-633de542:120c87e3365:-8000
```

Now, you are ready to add this node to the Resource Manager. Type:

```
rm-client -a rmi://hostname:port/MyNode
```

In order to have the right to use this command, you have to use the login and the password of an administrator. By default, an administrator whose login is **jl** and whose password is also **jl** is defined. After this operation, you can see the line:

```
Adding node 'rmi://hostname:port/MyNode' request sent to Resource Manager
```

This line indicates that your node has well been added to the RM. You can also check all the nodes of your Resource Manager by typing:

```
rm-client -ln
```

Finally, it just remains to launch the Scheduler as it was done in Section 4.1.5.1, “Launch the scheduler (easy way)”. The Scheduler will automatically look for an existing Resource Manager first. It will therefore find the one you have launched and use it. If you have launched the Resource Manager on a host different from those on which you want to start the Scheduler, you have to precise its URL to the 'scheduler-start' script.

There is also another way to start the Resource Manager directly with nodes using a deployment descriptor file. Please refer to the ProActive documentation at http://proactive.inria.fr/index.php?page=documentation to have more information about deployment/acquisition of nodes with the ProActive library. Such a deployment descriptor file is given in [Scheduler_Home_Dir]/config/rm/deployment/Local4JVMDeployment.xml. It is the file which is used when you proceed by the easy way. You can create your own deployment file and, then type the following line (with you own file) to start the RM:

```
rm-client -d ../../config/rm/deployment/Local4JVMDeployment.xml
```

This command is strictly equivalent to this one:

```
rm-client -localNodes
```

Note

The command **rm-client** can also be used in iteractive mode. Just type rm-client and then type '?' or 'help()' to learn how to use it in this mode.

4.1.5.3. Submit the job

Finally, submit your job using the **scheduler-client** script. This script can be used not only for submitting a job, but also, for instance, for retrieving a result (see Section 4.1.5.4, “Retrieving the result”). Type scheduler-client -h to find out all you can do wiht this script.

For submitting your job, type the following line:

```
scheduler-client -l user -submit ../../samples/jobs_descriptors/Job_nativ.xml
```

The **"-submit ../../samples/jobs_descriptors/Job_nativ.xml"** option specifies that you want to submit a job whose xml description file is ../../samples/jobs_descriptors/Job_nativ.xml. As for the **"-l user"** option, it specifies the login to use. By default, a login "user" exists and its password is "pwd". If no login is given to the script, a prompt will ask one and, in both cases (with or without specified login), the password will be asked to you.

Once done, you can see the line Job successfully submitted ! (id=1) which informs you that your job has well been submitted and gives you its ID (1 in this example). This ID will be used for retrieving the result of the job.

4.1.5.4. Retrieving the result

When the scheduling of a job is finished, scheduler stores its result and waits for the user to retrieve it. As it has already been said, the **scheduler-client** script is used to retrieve results. Launch on scheduler host this following command (still into the [Scheduler_Home_Dir]/bin/[OS]/ directory):

```
$ scheduler-client -l user -result ID
```

Type the user's password. The job result is then displayed:

```
Job 1 Result =>

task1: 0

task2: 0
```

You have two return codes of the native programs.

If you want to see the output of your job, you can use the **scheduler-client** script with the **output** option. Type:

```
$ scheduler-client -l user -output ID
```

Note

In the same way that for the command **rm-client**, you can also use the command **scheduler-client** in interactive mode. Just type scheduler-client and then type '?' or 'help()' to learn how to use it in this mode.

4.1.6. Using the GUI client application for job submission

An RCP graphical user interface exists for jobs submition and results consultation. Download the **Scheduler RCP Client** from http://www.activeeon.com/products and uncompress the application. Then, go to the Scheduler directory of uncompressed files, and launch the program:

```
$ ./Scheduler
```

You should have a window like this:

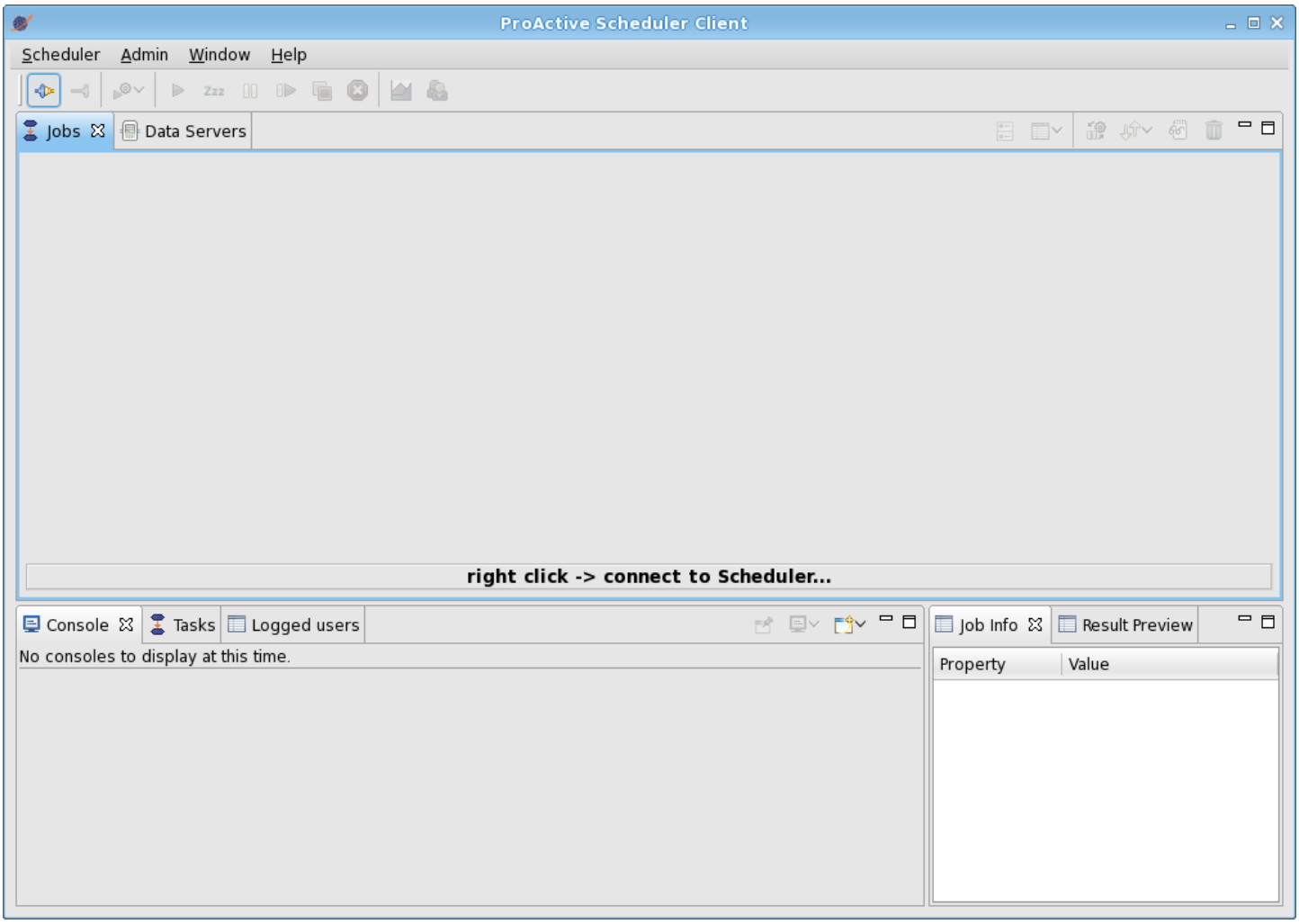


Figure 4.3. Scheduler GUI Client on startup

Right click to open connection dialog, and fill fields as follows:

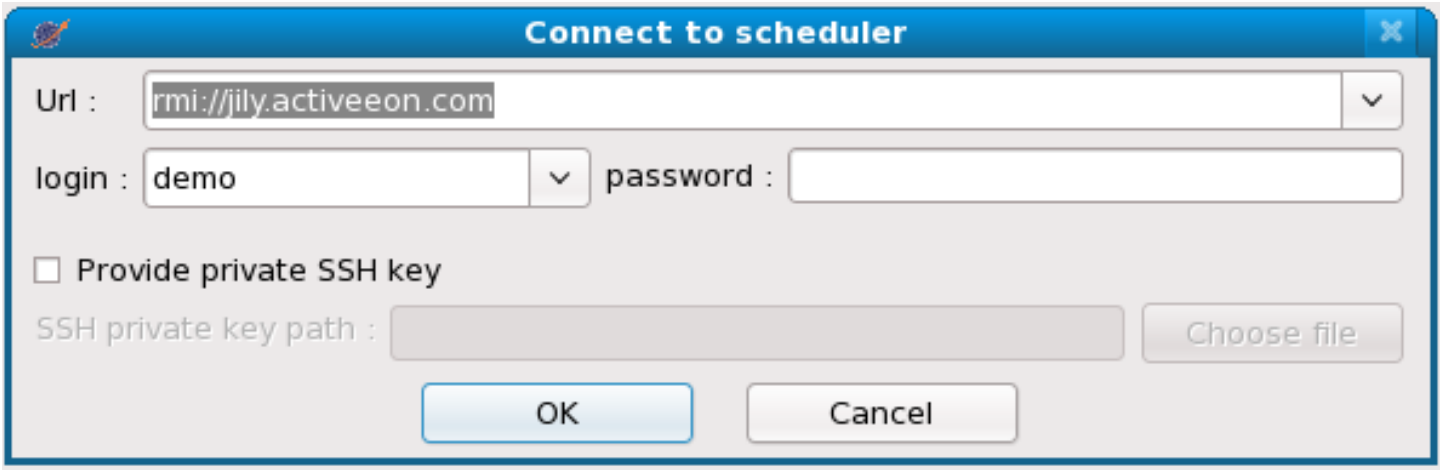


Figure 4.4. connection window of Scheduler GUI

Once connected to the Scheduler, you can see a first glimpse of the Scheduler activity. You have on the top left the list of pending jobs, on the top middle the list of currently running jobs with a progress bar, and on the top right the list of finished jobs which are waiting to be retrieved by their submitters.

To submit a new Job to the Scheduler, use the submit button in the toolbar, or activate the Scheduler menu and Submit submenu as pictured, the click 'Submit an XML Job file':

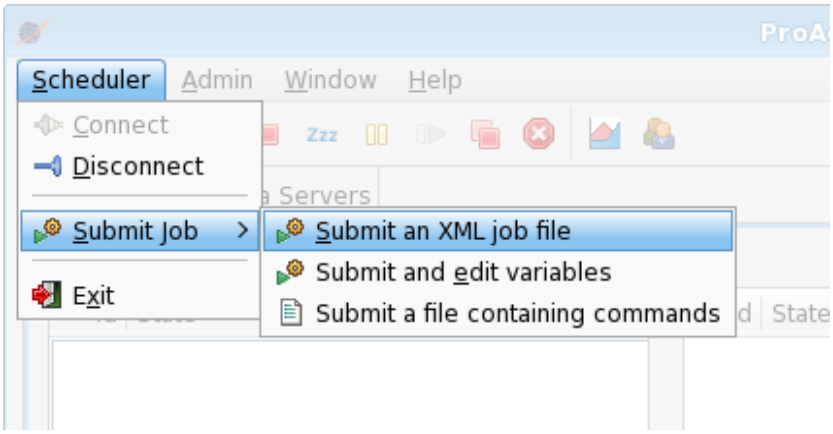


Figure 4.5. Context menu

This action creates a file explorer window. Go to your [working_dir], and double click on job_native.xml. The job is then submitted.

You can see your job staying in pending jobs area, which corresponds to the waiting queue. Then you can see your job's execution until it finally comes to the finished jobs list. Execution is then terminated. Note that you can submit several jobs at the same time.

If you click on the job line, you have at the bottom right of the interface, a description of the job, its execution time and a brief description. There is also in the task tab the list of tasks composing the jobs. Clicking on task's line, you will see on tab right below, the result obtained by the task.

4.2. Adding a selection script to the task

A very useful functionality is the possibility to add a selection script to a task. This selection script provides ability for the scheduler to find and select a node respecting criteria for the good execution of a task. Let's complete our previous nativ job example with a verification script, specifying that the executable nativTask must be executed on a Unix/Linux system.

4.2.1. XML job description

Complete your xml job descriptor as follows:

directory: [working_dir].

file: job_native.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
  name="job_nativ" priority="normal" cancelJobOnError="true"
  logFile="${EXEC_PATH}/nativTask.log">
  <description>Will execute 1 native C tasks</description>
  <variables>
```

```
    <variable name="EXEC_PATH" value="[working_dir]"/>
  </variables>
<taskFlow>
    <task name="task1" preciousResult="true">
      <description>Will display 10 dots every 1s</description>
      <selection>
        <script>
        <file path="${EXEC_PATH}/check_unix_os.js"/>
        </script>
      </selection>
      <nativeExecutable>
        <staticCommand value="${EXEC_PATH}/nativTask">
          <arguments>
            <argument value="1"/>
          </arguments>
        </staticCommand>
      </nativeExecutable>
    </task>
  </taskFlow>
</job>
```

We can see in the two tasks definitions, after the task's description, that there is a new “selection” tag which contains a path to a Javascript file. When the task is executed, the scheduler will launch this script to nodes, and will select a node that has answered "yes" at the execution of this script and then will deploy the task to this node.

4.2.2. Code of the node selection Javascript

Create in your [working_dir] a file as follows:

directory: [working_dir].

file: check_unix_os.js

```
importPackage(java.lang);

if(System.getProperty("os.name").contains("Windows"))
   selected=false;
else
   selected=true;
```

There is just one rule to know for verifying scripts creation: the script has to contain a variable named "**selected**" which has to be setted to true or false at the end of script execution. Node selection mechanism is simple: Scheduler asks to Resource Manager one or several nodes that verify that script. So Resource Manager will execute this script on its nodes. If after execution, one of its node has its "selected" variable set to true, Resource manager considers that this node verify selection script and gives node to scheduler which will execute the task on it. If a node return "selected" variable set to false, Resource manager will try to provide scheduler with another node which validates this selection script.

4.3. PreScript and PostScript

Another functionality is the possibility to define pre and post scripts. For a given task (Java task or native task), it is possible to launch a script before and after its execution. This possibility can be useful to copy files to a node, or clean a directory before or after task execution, for example. This is a way to separate from business code the preparation of execution environment and its cleaning. This section proposes you to add to your native job example a script which removes a list of files from a specified directory.

4.3.1. XML job description

Complete your xml job descriptor as follows:

directory: [working_dir]

file: job_native.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
  name="job_nativ" priority="normal" cancelJobOnError="true"
  logFile="${EXEC_PATH}/nativTask.log">
  <description>Will execute 1 native C tasks</description>
  <variables>
    <variable name="EXEC_PATH" value="[working_dir]"/>
  </variables>
<taskFlow>
    <task name="task1" preciousResult="true">
      <description>Will display 10 dots every 1s</description>
      <pre>
        <script>
          <file path="${EXEC_PATH}/remove_files.js">
            <arguments>
              <argument value="${EXEC_PATH}/1.tmp" />
            </arguments>
          </file>
        </script>
      </pre>
      <nativeExecutable>
        <staticCommand value="${EXEC_PATH}/nativTask">
          <arguments>
            <argument value="1"/>
          </arguments>
        </staticCommand>
      </nativeExecutable>
      <post>
        <script>
          <file path="${EXEC_PATH}/remove_files.js">
            <arguments>
              <argument value="${EXEC_PATH}/2.tmp" />
            </arguments>
          </file>
        </script>
      </post>
    </task>
  </taskFlow>
</job>
```

4.3.2. Code of the removing files Javascript

Create in your [working_dir] a file as follows:

directory: [working_dir]

file: remove_files.js

```
importPackage(java.io);
print("clean working directory \n");
for(i=0; i<args.length;i++)
{
   var f= new File(args[i]);
   if(f["delete"]()) {
      print(args[i] +" deleted\n");
   } else {
      print("deleting "+ args[i] +" failed\n");
      }
}
```

Create in your [working_dir] two files 1.tmp and 2.tmp, and execute the job. You will see that 1.tmp is removed just before task starting, and 2.tmp is removed at task's end.

4.3.3. File Transfer Helpers to be used from scripts

The package **org.ow2.proactive.scripting.helper.filetransfer** provides several file transfer drivers that can be used in the pre and post scripts. SCP, SFTP and FTP protocols are implemented. You just have to specify, in the script code, the protocol to be used and, optionally, the implementation (if several implementations are available for the given protocol).

Here is an example of a script that copies a set of files from a remote host to a local folder.

file: getFiles.js

```
importPackage(java.lang);
importPackage(java.io);
importPackage(java.util);
importPackage(org.ow2.proactive.scripting.helper.filetransfer);
importPackage(org.ow2.proactive.scripting.helper.filetransfer.driver);
importPackage(org.ow2.proactive.scripting.helper.filetransfer.initializer);

//This script example ilustrates the use of org.objectweb.proactive.scheduler.helper API to get files from the remote host
 to the local node

//set this to true if yo want to have your logs in a file on the remote machine
var logToFile = false;
//set this to true to enable debug mode
var mode_debug= false;

var task_id=System.getProperty("pas.task.id");
var logsFile = "task_"+task_id+".log";


log ("Start Get Files Script");
//only if logToFile is true:
log("logs file at "+System.getProperty("java.io.tmpdir")+File.separator+logsFile);
```

```
//-------------- Script ARGS --------------------------
  if (args.length<5)
   {
     log("Script usage: host username password working_dir_on_node file1 [file2] ... [filen]");
     log("Not enough parameters. Script cannot be executed");
     throw new Exception ("Not enough parameters.");
   }


     //host where the file is to be copied from
     var host = args[0];
     debug("remote host="+host);

     //identification to the remote host
     var username=args[1];
     debug("user="+username);
     var password=args[2];
     debug("password="+password);
     //for the scp implementation- the driver will first try to connect through ssh keys
     //and will use the password only if the connection fails.

     var working_dir_on_node = args[3];
     debug("destination folder on the compute node: "+working_dir_on_node);

     debug("Files to copy: ");
     var files = new LinkedList();

     var k=0;
     for (k=4;k<args.length;k++)
        {
        files.add(args[k]);
        debug(args[k]);
        }


     //how long will it take?
     var t1=System.currentTimeMillis();


     //-------- Create Local Folder if it doesn't exist
     var tFolder = new File(working_dir_on_node);
     if (!tFolder.isDirectory())
     if (tFolder.mkdirs())
     {
        debug ("Folder have been created: "+tFolder);
     }
     else
     {
       debug("Could not create folder " + tFolder );
     }
```

```
 //---------- Define Driver to be used for file transfer (or use default one) ----------


//----(SCP Protocol) Use this code for in order to use SCP_Trilead_Driver to copy the files ----
//(By default, the initializaer will use SCP_Trilead_Driver to init the connection)
var ftInit= new FileTransfertInitializerSCP(host, username, password);

// ---------- OR ----------------



//----(SFTP Protocol) Use this code for in order to use SFTP_Trilead_Driver to copy the files ----
//    var driver = new SFTP_Trilead_Driver();
//    var ftInit= new FileTransfertInitializerSCP(host, username, password,driver.getClass());

// ---------- OR ----------------
//---- (SFTP Protocol) Use this code for in order to use SFTP_VFS_Driver to copy the files ----
//        var driver = new SFTP_VFS_Driver();
//        var ftInit= new FileTransfertInitializerSCP(host, username, password,driver.getClass());


// ---------- OR ----------------
//---- (FTP Protocol) Use this code for in order to use FTP_VFS_Driver to copy the files ----


//        var driver = new FTP_VFS_Driver();
//        var ftInit= new FileTransfertInitializerFTP(host, username, password,driver.getClass());


    // ---------- Create  session for the file transfer ----------
    var session = new FileTransfertSession(ftInit);

    // ---------- Copy the files ----------
    session.getFiles(files, working_dir_on_node);


    var t2=System.currentTimeMillis();
    log("Get files script ended. Copying files took "+(t2-t1)/1000+" seconds = "+(t2-t1)/60000)+" minutes";



function log(msg)
{
      msg="(getFiles.js) "+msg;
       if (logToFile)
           {
               ScriptLoggerHelper.logToFile(logsFile,"\n"+msg+"\n");
           }

       println(msg);
}
```

```
function debug(msg)
{
      if (mode_debug)
      {
       log ("DEBUG: "+msg);
      }
}
```

The commented lines in the "Define Driver to be used for file transfer" section in the script shows how one can choose between different protocols and implementations.

This script, as well as a putFiles.js script can be found in the samples/scripts/filetransfer folder of the Scheduler Project.

4.4. Command generator script

You have seen that a native task can be launched by the scheduler on different operating systems, unix or Windows for example. That is because Resource Manager can handle nodes from different computer architectures. You have also seen the possibility to select nodes compatible with the native task to execute. But you can also adapt the native command to execute, corresponding to the node that Resource manager has provided. ProActive Scheduler provides the possibility to dynamically generate the native command to launch for a task. This functionality is convenient if you have versions of your native program for different OS, or different versions of the native executable, optimized for dynamic libraries which differ from a host to another.

This section shows you how to develop a job with a task able to launch our native executable “nativTask” on windows system or on a UNIX system. First compile and build nativTask.c on a windows system in order to have a Windows specific version. Let's implement a task with a command generator.

4.4.1. XML job description

Complete your xml job descriptor as below:

directory: [working_dir].

file: job_native.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
   name="job_nativ" priority="normal" cancelJobOnError="true"
   logFile="${EXEC_PATH}/nativTask.log">
   <description>Will execute 1 native C task with command generation</description>
   <variables>
      <variable name="EXEC_PATH" value="[working_dir]"/>
   </variables>
   <taskFlow>
      <task name="task1" preciousResult="true">
      <nativeExecutable>
         <dynamicCommand>
            <generation>
               <script>
```

```
                <file path="${EXEC_PATH}/commandGenerator.js"/>
              </script>
            </generation>
          </dynamicCommand>
        </nativeExecutable>
        </task>
    </taskFlow>
</job>
```

4.4.2. Code of the command generator script

Create in your [working_dir] a file as follows:

directory: [working_dir].

file: commandGenerator.js

```
importPackage(java.lang);

if(System.getProperty("os.name").contains("Windows"))
command="c:\nativTask.exe"
else
command="[working_dir]/nativTask"
```

The script gets the OS type and build a command corresponding to it. A generation script has to define a variable named “command” which contains the native command to execute. That is the only rule to respect. Then, scheduler will get this variable and execute the native command defined by the Javascript.

4.5. Using exported environment variables

Each job has a specified name in its deployment descriptor as well as a unique job ID given at the moment of job's submission. All tasks have these two same parameters specified at the same time. You can access to these values in task's execution environment, during task's execution. When a native task is launched, the task has also 4 exported environment variables:

- **$PAS_JOB_NAME** - Job's name defined in its XML descriptor
- **$PAS_JOB_ID** - unique job's ID given at submission time.
- **$PAS_TASK_NAME** - name of the task currently launched (name defined in XML descriptor).
- **$PAS_TASK_ID** - unique task's ID.

Now you will submit a native executable which produces a file in output. The following example launches in parallel 4 executions of a native C executable which takes an integer in parameter and produces a file named output.txt in its current (launching) directory. If you launch simultaneously several executions instances, it will produce a collision in file output, because the different execution instances will create and write into the same file in the same directory. As you don't want to be annoyed by creation of directories for each execution instance, this job provides a launcher.sh Unix shell which performs these operations automatically. launcher.sh creates before launching the executable, a temporary dir for each task, its name corresponds to current task's ID environment exported variable. So this job is made of a XMl job descriptor, a native executable, and shell script which performs the launching.

4.5.1. XML job description

Create a new xml job descriptor and complete as follows:

directory: [working_dir].

file: job_file_output.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:dev http://www.activeeon.com/public_content/schemas/
proactive/jobdescriptor/dev/schedulerjob.xsd"
  name="job_file_output" priority="normal" cancelJobOnError="false"
  logFile="${WORK_DIR}/job_file_output.log" >
  <description>Will execute a 4 native C tasks, avoiding file output collision</description>
  <variables>
    <variable name="EXEC_PATH" value="[working_dir]"/>
  </variables>
  <taskFlow>
    <task name="Native_task_file_output_1" preciousResult="true">
      <description>Will display 10 dots every 2s an log in an output.txt file</description>
      <nativeExecutable>
        <staticCommand value="${EXEC_PATH}/launcher.sh">
          <arguments>
            <!-- native executable to launch -->
            <argument value="${EXEC_PATH}/task_with_output"/>
            <!--  finally, effective arguments list for the executable -->
            <argument value="3"/>
          </arguments>
        </staticCommand>
      </nativeExecutable>
    </task>
    <task name="Native_task_file_output_2" preciousResult="true">
      <description>Will display 10 dots every 4s an log in an output.txt file</description>
      <nativeExecutable>
        <staticCommand value="${EXEC_PATH}/launcher.sh">
          <arguments>
            <argument value="${EXEC_PATH}/task_with_output"/>
            <argument value="4"/>
          </arguments>
        </staticCommand>
      </nativeExecutable>
    </task>
    <task name="Native_task_file_output_3" preciousResult="true">
      <description>Will display 10 dots every 1s an log in an output.txt file</description>
      <nativeExecutable>
        <staticCommand value="${EXEC_PATH}/launcher.sh">
          <arguments>
            <argument value="${EXEC_PATH}/task_with_output"/>
            <argument value="1"/>
          </arguments>
        </staticCommand>
      </nativeExecutable>
    </task>
    <task name="Native_task_file_output_4" preciousResult="true">
      <description>Will display 10 dots every 1s an log in an output.txt file</description>
      <nativeExecutable>
        <staticCommand value="${EXEC_PATH}/launcher.sh">
          <arguments>
            <argument value="${EXEC_PATH}/task_with_output"/>
            <argument value="1"/>
          </arguments>
```

```
            </staticCommand>
          </nativeExecutable>
      </task>
   </taskFlow>
</job>
```

4.5.2. Native C code of the executable which produces an output file

Create your native C executable with the code below:

project: ProActive

directory: descriptors/deployments/scheduler/jobs/job_native_linux_file_output

file: nativTask.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

#define DEFAULT_WAIT_TIME 1

int main(int argc, char**argv){

   int i;
   FILE* fd;

      fd = fopen("output.txt", "w+");
      chmod("fichier.txt", 0666);
   printf("Starting native task...\n");

   /* parsing args */
   int nb = DEFAULT_WAIT_TIME;
   if(argc > 1){
      printf("argv[1]= %s\n", argv[1]);
      sscanf(argv[1], "%d", &nb);
      printf("Waiting time : %d s\n", nb);
   }

   fprintf(fd, "Execution started, waiting parameter is : %d \n", nb);
   for(i = 0; i < 10; i++){
      printf(".");
      fprintf(fd, ".");
      fflush(stdout);
      sleep(nb);
   }

   fprintf(fd, "\nEnd of execution\n");
   fclose(fd);
   printf("\nNative task terminated !\n");
   return EXIT_SUCCESS;
}
```

You have also a unix compiled version of this program, named task_with_output, in the same directory.

4.5.3. Launching shell script

Finally, write your launching Shell:

project: ProActive

directory: descriptors/deployments/scheduler/jobs/job_native_linux_file_output

file: launcher.sh

```
#!/bin/sh

# this shell launches a native command $1 from a temporary directory
# nammed by $PAS_TASK_ID env variable specified by ProActiveScheduler
# at Job submission

if [ $# -lt 1 ]
then
   echo "you must specify at least the command to launch"
   echo "Usage: launcher.sh executable_to_launch  [parameters...]"
   exit 1
fi

cd $(dirname $0)

COMMAND=$1
shift

if [ -z $PAS_TASK_ID ]
then
   echo "Error : environment var \$PAS_TASK_ID is not defined,"
   echo "Execution aborted"
   exit 1
fi

if [ -e $PAS_TASK_ID ]
then
   echo "directory path exists !"
   echo "Execution aborted"
   exit 1
fi

mkdir $PAS_TASK_ID
if [ $? -ne 0 ]
then
   echo "directory $PAS_TASK_ID creation failed"
   echo "Execution aborted"
   exit 1
fi

cd $PAS_TASK_ID
$COMMAND $@
if [ $? == 0 ]
then
   echo "execution of $COMMAND ok"
```

```
else
    echo  "problem during execution of $COMMAND $@"
fi
```

Copy your compiled executable (task_with_output) and your launching shell script, and submit this job. You will see, after job's execution, 4 directories created in your [working_dir] named by the 4 tasks ID of your execution, and directories contains a file, output.txt, corresponding to outputs of the 4 execution instances launched by the job.

Chapter 5. ProActive Scheduler Eclipse Plugin

Scheduler Eclipse Plugin is a **graphical client** for remote monitoring and control of ProActive Scheduler (see Chapter 2, *User guide*), including remote submission of XML-defined jobs (see Section 2.1.1, "Job XML descriptor").

The Scheduler Graphical Client is a standalone application built on top of Eclipse Rich Client Platform (RCP) [1]. It is available for all major platforms (Windows, Mac OS, Linux, Solaris).

Although the plugin is built as a standalone application, it can be integrated in an existing Eclipse installation by installing only the Scheduler Plugin, allowing the use of the Scheduler Perspective.

5.1. Starting Scheduler Graphical User Interface

To start the ProActive Scheduler Client, run the executable named **Scheduler** (Unix) or **Scheduler.exe** (Windows).

Once started, the main window of the client should look like this:

[1] http://wiki.eclipse.org/index.php/Rich_Client_Platform

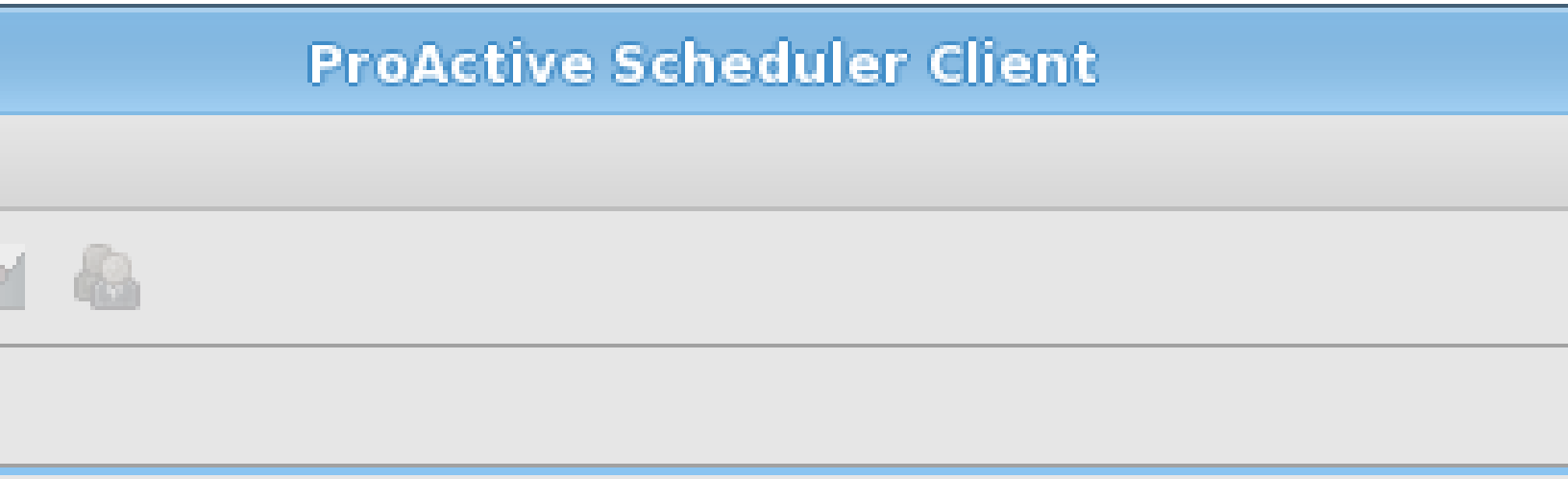


Figure 5.1. Default Scheduler perspective

5.2. Connect to an existing Scheduler

A Scheduler has to be already started before connecting the Scheduler GUI. If no scheduler has been started, please refer to Section 1.2, “Scheduler Installation”.

To connect to an existing Scheduler, bring up the connection dialog by clicking the first icon in the toolbar, or its menu entry in the 'Scheduler' menu. Note that the Scheduler Connection dialog should be showed automatically when starting the application. "Connect the ProActive Scheduler" in the context menu as shown below:

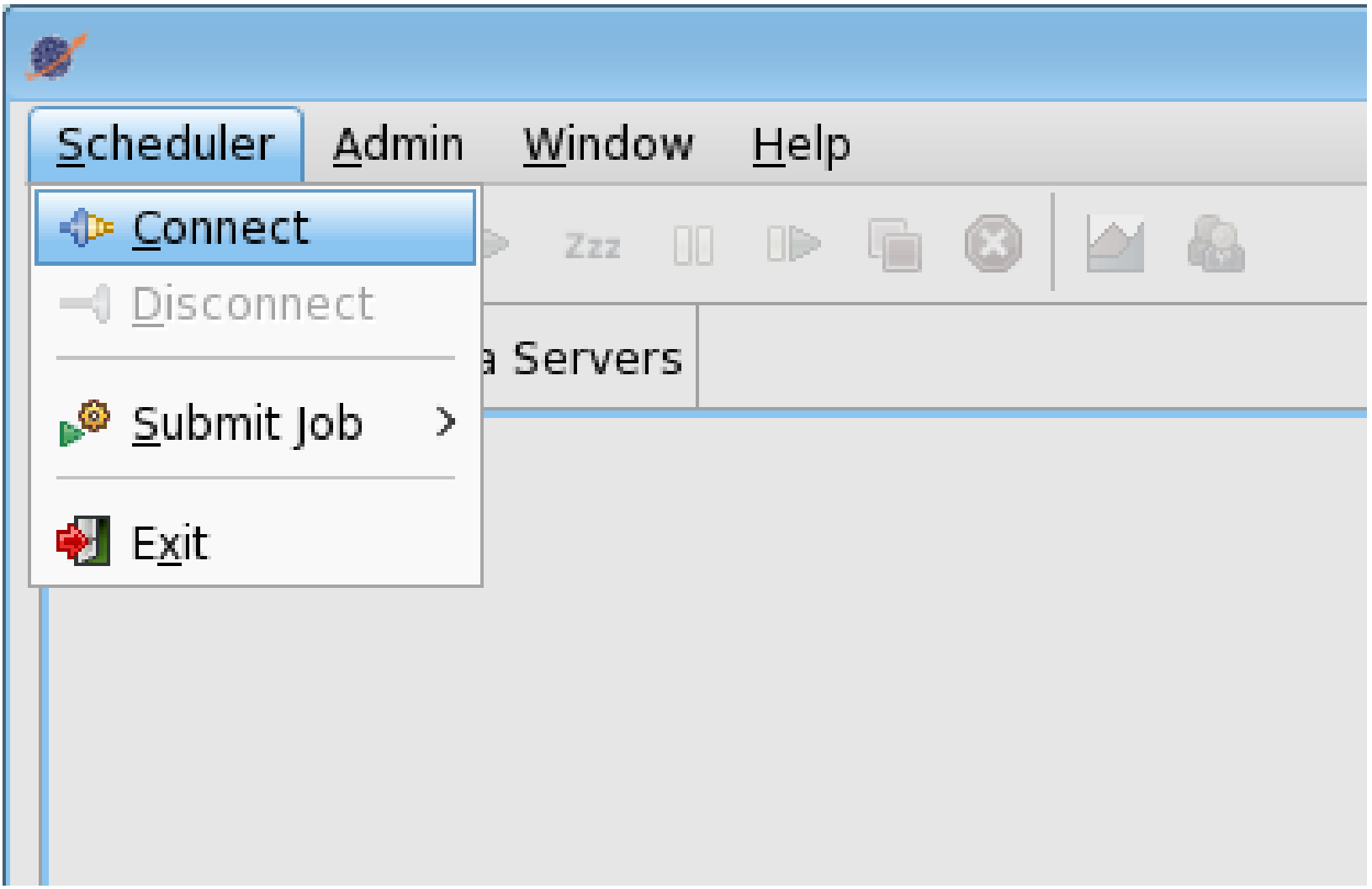


Figure 5.2. Scheduler connection (1)

Some information are now requested in order to establish the connection:

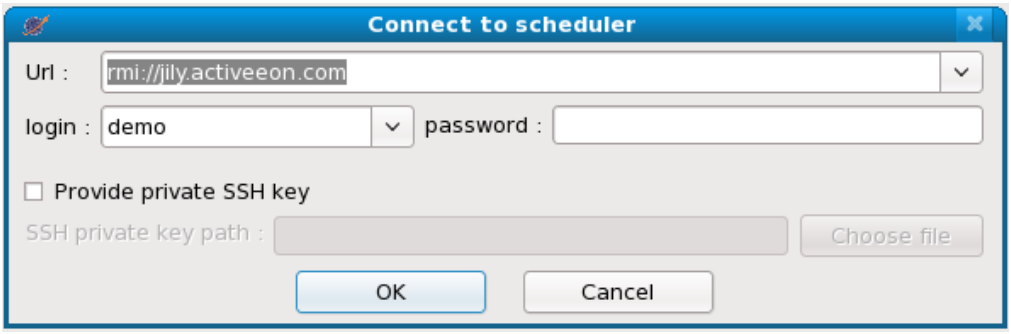


Figure 5.3. Connection dialog

Enter required information about the remote scheduler and click on **OK**. Url is made of the protocol, the host name on which the Scheduler is started and the port on which it is listening. Then, enter your user/admin name and password.

5.3. Scheduler perspective

Scheduler plugin provides the **Scheduler perspective** [2] displayed in the Figure 5.4, “Scheduler Perspective”.

[2] http://help.eclipse.org/help31/index.jsp?topic=/org.eclipse.platform.doc.user/gettingStarted/qs-43.htm

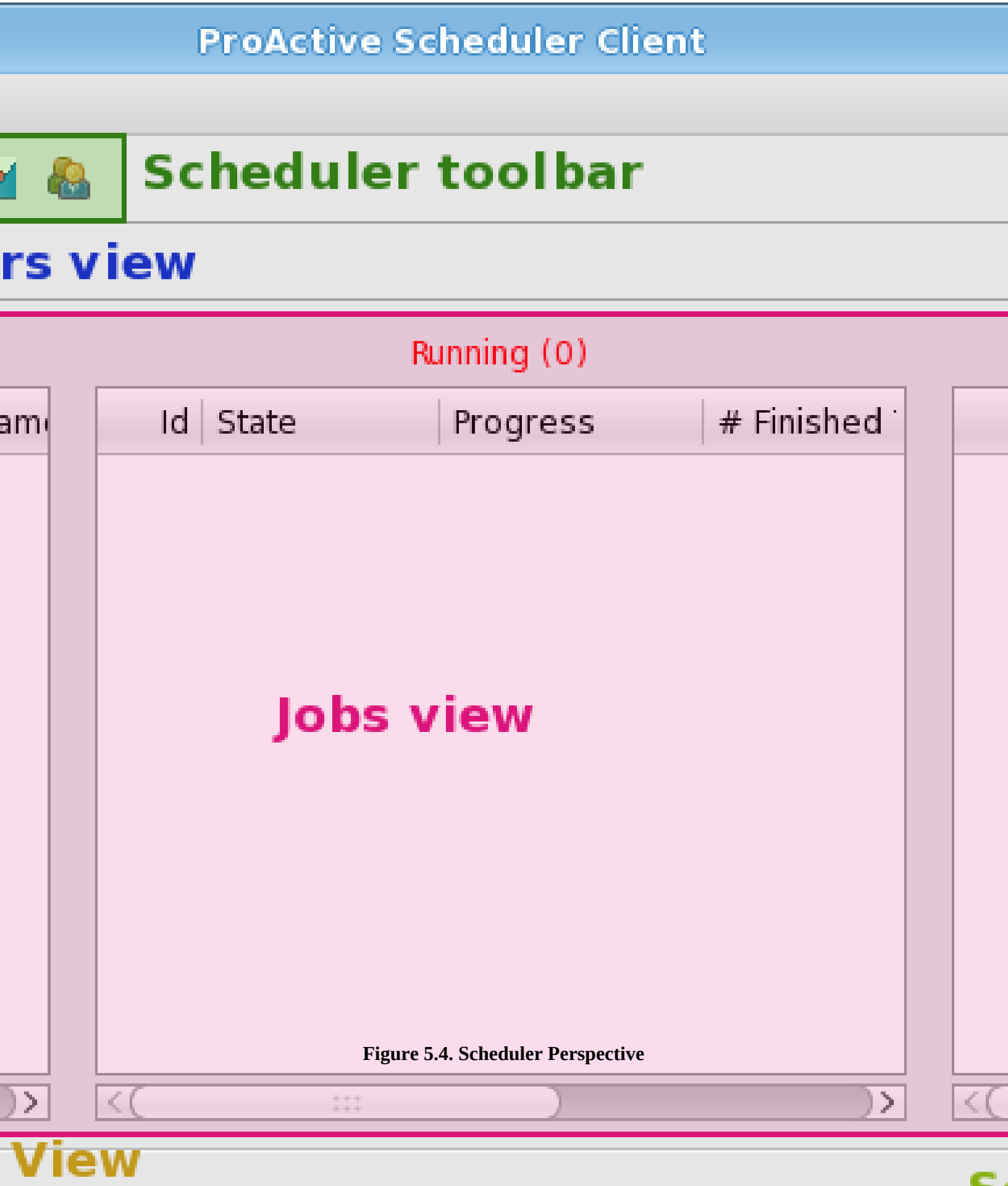


Figure 5.4. Scheduler Perspective

This perspective defines the following set of views [3]:

- **Jobs** view - shows pending, running and finished jobs in the scheduler, and interact with them.
- **Data Servers** view - start, stop, add or remove Data Servers to use for submitted jobs.
- **Console** view - shows jobs standard and error output (on demand).
- **Tasks** view - displays detailed informations on tasks contained in the selected job.
- **Logged Users** view - displays information about all users connected to the Scheduler this client is connected to.
- **Jobs Info** view - displays all information of the selected job.
- **Result Preview** view - displays a textual or graphical preview on the selected task result. A simple click will show the internal view while the double click start an external result preview (as a pictures editor or viewer) if provided.
- A **Scheduler toolbar** - perform tasks relative to the entire application, or the connected scheduler itself: connection, job submission, scheduler administration, statistics and accounting.

5.4. Views composing the perspective

5.4.1. Jobs view

The main view is obviously the jobs one:

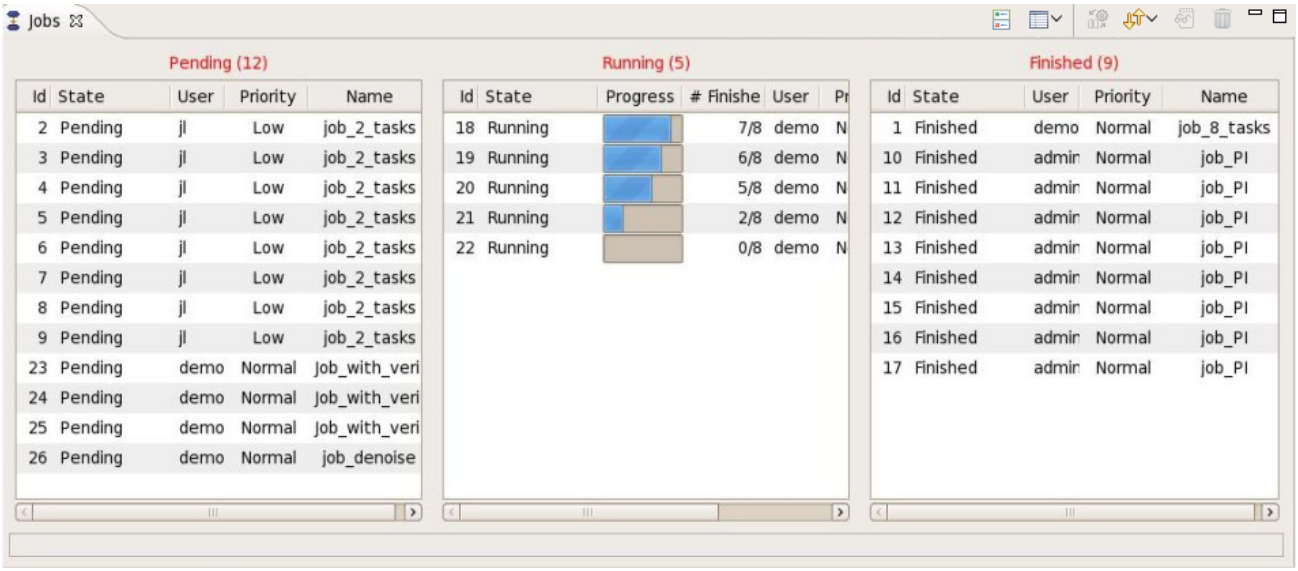


Figure 5.5. Jobs view

This view is composed of 3 panels that represent respectively pending, running and finished jobs. In each panel you can watch many different information about jobs, like their states, names, IDs, progress... It is also possible to switch between vertical panel presentation and horizontal panel presentation as it will be described later.

In each panel, the list of jobs can be ordered by column.

The contextual menu of the view, available when right-clicking on a selected task, enables the following actions:

- **Pause/Resume a Job** - use this button to pause a pending or running job, and to resume a paused job. Note that it is not possible to pause a running task. So once a job paused, each running task will be finished. If a task has to restart for any reasons, it will be paused instead of restarted.

[3] http://wiki.eclipse.org/index.php/FAQ_What_is_a_view%3F

- **Select Job priority** - use this button to change the priority of your job. Allowed priorities for a user are:
  - Normal
  - Low
  - Lowest
- **View selected job output** - use this button to view the selected job output. Job output is in fact the output that comes from the different tasks from both standard output and standard error stream.
- **Kill the selected job** - use this button to kill a job. The job will be terminated immediately, removed from the queue where it was, and its results will be lost.

5.4.2. Data Servers view

The Data Server view enables users to start new instances of PADataServer that can mount local filesystems as a DataSpaces object usable among all computation nodes.

Multiple servers can be defined, and not all servers must be running at the same time: they can be started or stopped at will. When exiting the client and restarting it, all defined servers will be saved and automatically added in a stopped state.

To create a new Server, use the contextual menu (right-click) or the view's iconbar:

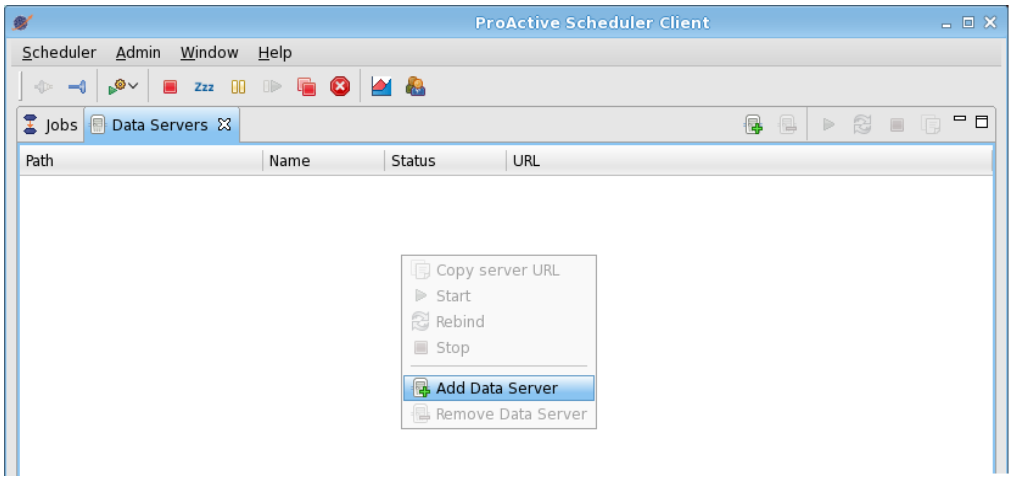


Figure 5.6. Default Data Server view

Creation of a server requires the following information:

- Root directory: the directory on the local filesystem to use as the server root
- Name: an unique identifier for the server to start
- Rebind: if a server with the same name was already started outside the Scheduler Client, creation of the server will fail. Using the rebind option, the client will reclaim this already created server.
- Start server: either add the server in stopped or started mode.

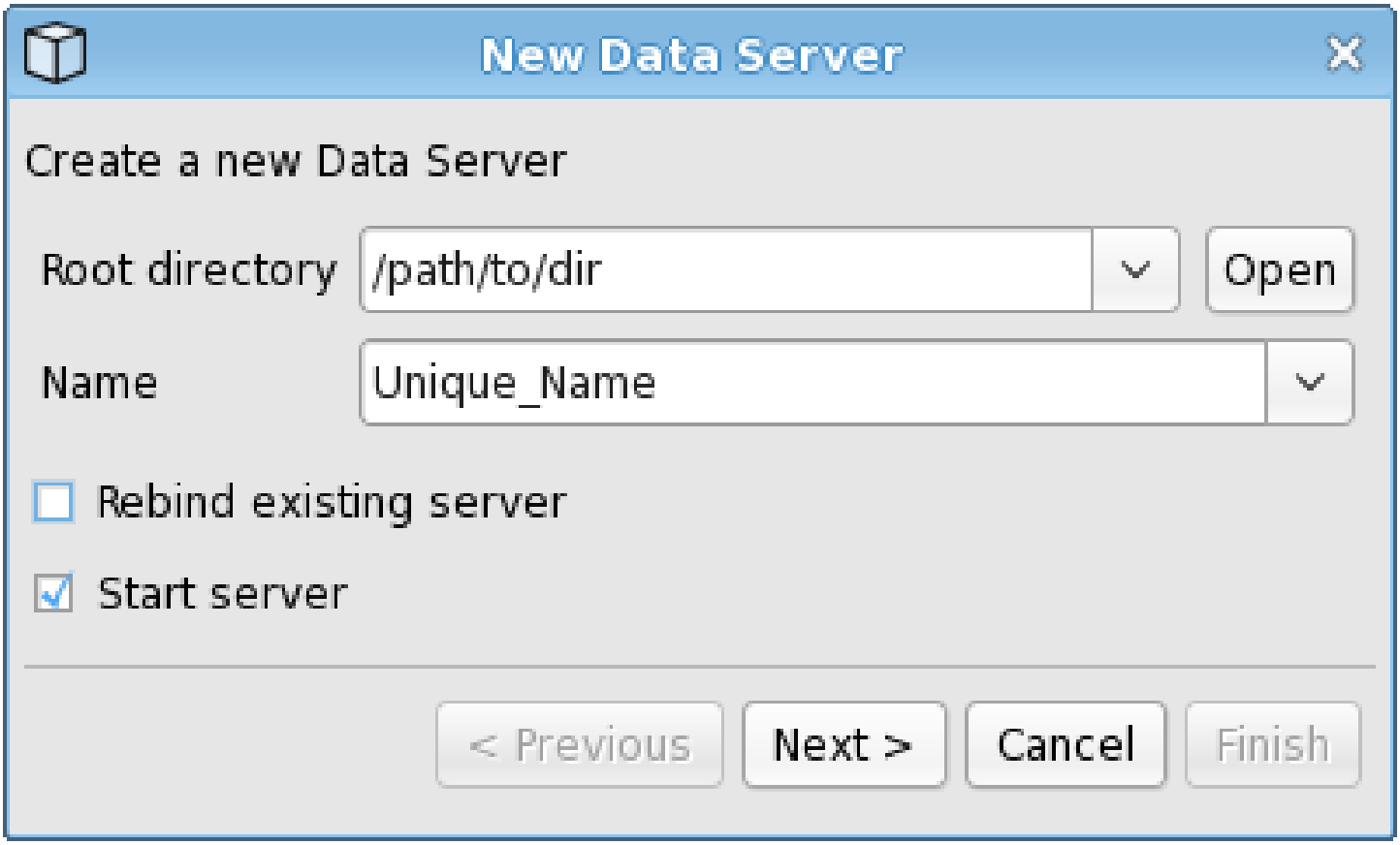


Figure 5.7. Create a new Server

Once a server has been added to the list, users can interact with it using the contextual menu by right-clicking on the requested server. The server can be started or stopped depending its state, or even removed definitely from the list.

After you servers have been added and started, they can be used when submitting new jobs. See Section 5.4.8, “Submit XML Job and edit variables”.

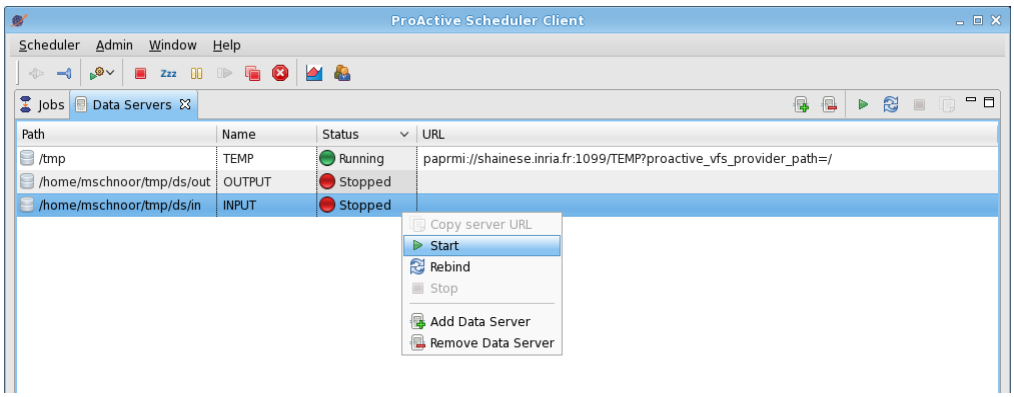


Figure 5.8. Interaction with the defined servers

5.4.3. Console view

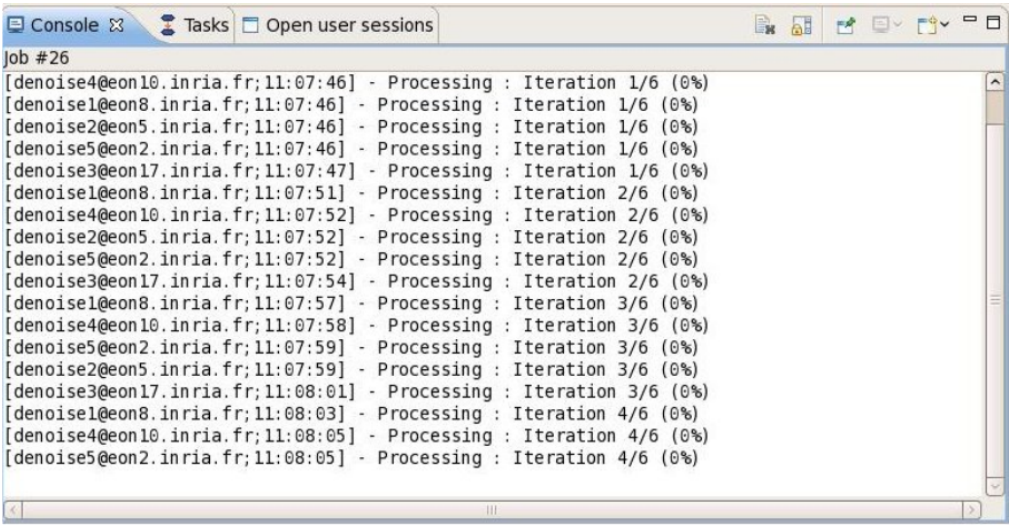


Figure 5.9. Console view

This view displays selected job standard and error output (only on demand).

5.4.4. Tasks view

Figure 5.10. Tasks view

This view provides a lot of information on every tasks composing a job like:

- **Id** - task id generated by the Scheduler
- **State** - task status to know if it is finished, running, etc...
- **Name** - task name given by the user
- **Host name** - host name on which the task is executed
- **Start time** - task start time
- **Finished time** - task finished time
- **Re-run** - In the column "Re-run", the first number represents how many times the task **has already been** re-executed, and the second number represents how many times the task **can be** re-executed

- **Description** - The user description of the task

In this view, the list of tasks can be ordered by column.

5.4.5. Job Info view

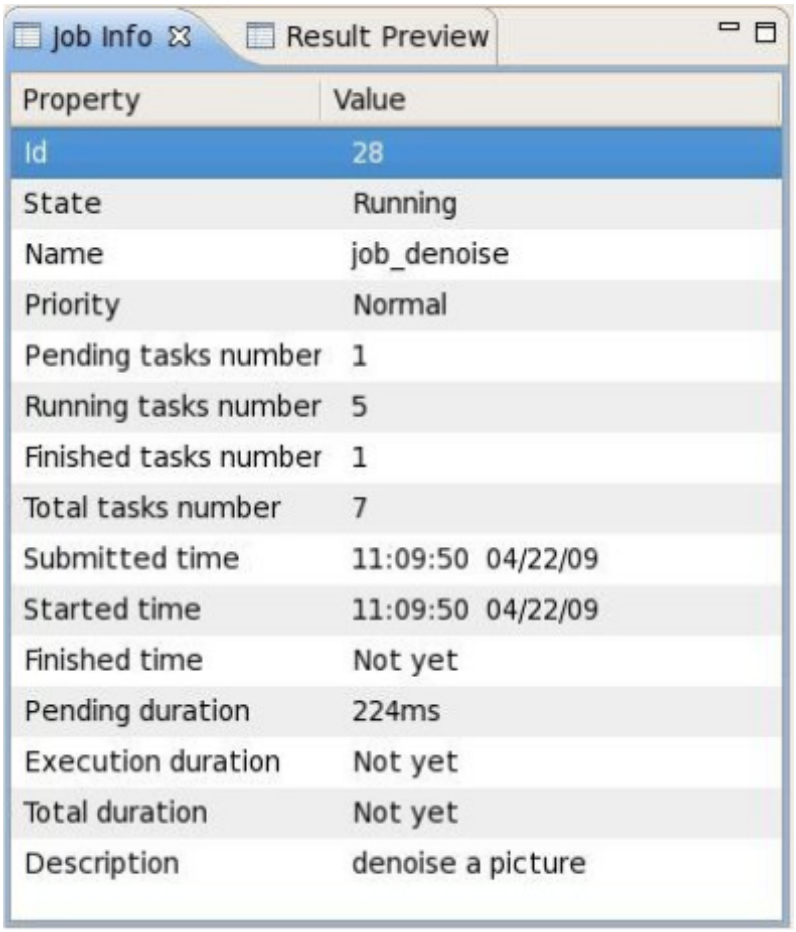


Figure 5.11. Job Info view

This view provides many informations on the selected job like:

- **Id** - job id generated by the Scheduler, can be useful to get the result of your job outside of the GUI.
- **State** - job status to know if it is pending, running, paused...
- **Name** - job name given by the user that has submitted this job
- **Priority** - current job priority (this can be changed to accelerate its execution)
- **Pending tasks number** - number of pending tasks
- **Running tasks number** - number of running tasks
- **Finished tasks number** - number of finished tasks
- **Total tasks number** - total number of tasks composing the job
- **Submitted time** - job submitted time
- **Started time** - job start time
- **Finished time** - job finished time

- **Description** - user description of the job

5.4.6. Result Preview

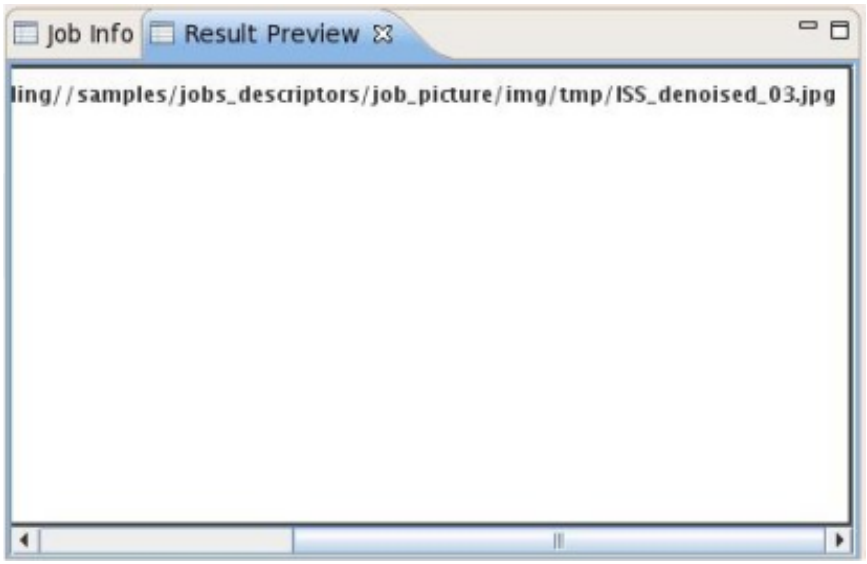


Figure 5.12. Result Preview

This view displays result of the selected task (selected in task view), according to the ResultPreview field. This preview is powerful because of the generic way that will be used to perform the display. It is possible to redefine the displaying behavior by implementing the **ResultPreview** abstract class. Using this mechanism, it is possible to use an external tool or soft to display the result like an image viewer for picture (see Section 2.2.3, “Tasks options and explanations” for details). Here is an example of what can be done:

Figure 5.13. custom Result Preview

5.4.7. Scheduler control panel

This part explains how to use the control panel available at the top right of the Scheduler. The control panel goal is to manage the different jobs and the Scheduler itself. Let's start with the possibilities offered to a user:

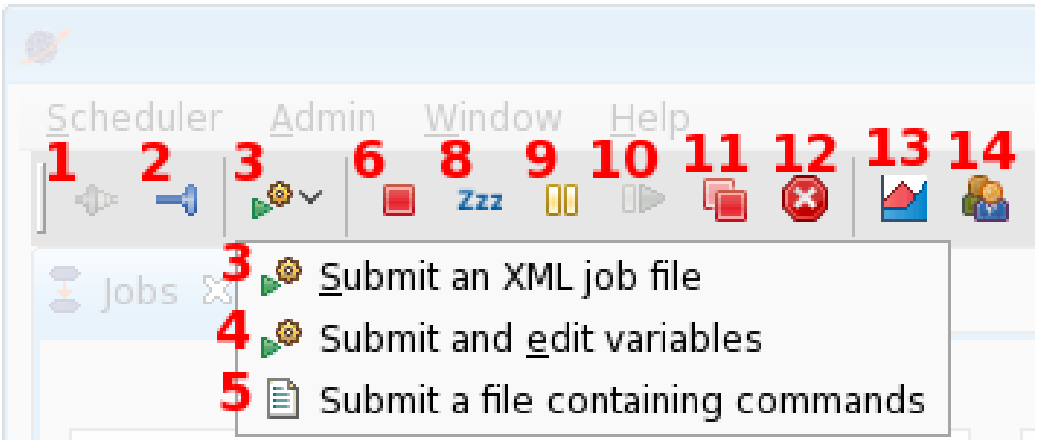


Figure 5.14. Control Panel

5.4.7.1. User controls

- **(1)(2) Connect/Disconnect the Scheduler** - use this button to display the "Connect to scheduler" dialog in order to establish a connection to a remote ProActive Scheduler. Once connected this button disconnects the user from the current connected Scheduler.
- **(3) Submit a Job from an XML file** - use this button to submit a new job to the Scheduler. This action will prompt a file chooser dialog so that you can select one or several XML job descriptors. If you don't know how to create such a descriptor, please refer to Section 2.1.1, “Job XML descriptor”.
- **(4) Submit a Job from an XML file, edit variables definition** - This button has the same effect as the previous one, with the difference that after an XML job descriptor is selected for submission, a new dialog appears to edit the definitions of the job's variables. See Section 5.4.8, “Submit XML Job and edit variables”.
- **(5) Submit a file containing native commands** - use this button to submit a job defined in flat file. If you don't know how to create such a job descriptor, please refer to Section 2.1.5, “Create a job from a simple flat file”. This button opens a wizard for submitting this kind of job (see section Section 5.4.9, “Command file job submission”).
- **(13)** Runtime monitoring - Show detailed monitoring information about the scheduler activity: pending, running, finished jobs, users.
- **(14)** My Account - Show information about the user account currently logged to the scheduler.

All these actions that can be performed in the control panel can be also executed from the right click context menu. To view the result of your task, select one in the tasks view and the result preview will be displayed automatically.

5.4.7.2. Administrator controls

All actions allowed in user mode are also allowed in Administrator mode. Moreover, you can execute any action on any job even you are not the owner of this job. It is also possible for an administrator to do the following operations:

- **(6) Stop the Scheduler** - This action places the Scheduler in a mode where no job can be submitted anymore until a restart. Nevertheless, all running tasks will be terminated.
- **(7) Freeze the Scheduler** - The behavior is the same as those of the stop button except that job can still be submitted.
- **(9) Pause the Scheduler** - This action put the Scheduler in a Pause status. Job can be submitted again but no job will go to the running queue. Every running job will terminate.

- **(10) Resume the Scheduler** - Put the Scheduler to the normal scheduling mode, that is, to the Started mode. This action can be performed when the scheduler is Frozen or Paused.
- **(11) Shutdown the Scheduler** - Put the Scheduler in its shutdown sequence. Submitting job is no longer available. It will wait for every pending and running job to be terminated. Once done, the Scheduler will shutdown and disconnect.
- **(12) Kill the Scheduler** - Force the Scheduler to terminate immediately. This action kills every task and job, get the resources back to the Resource Manager and kill the Scheduler.

5.4.8. Submit XML Job and edit variables

In addition to the regular submission button, a second job submission method allows users to edit the variable definitions of an XML job descriptor, prior to submission.

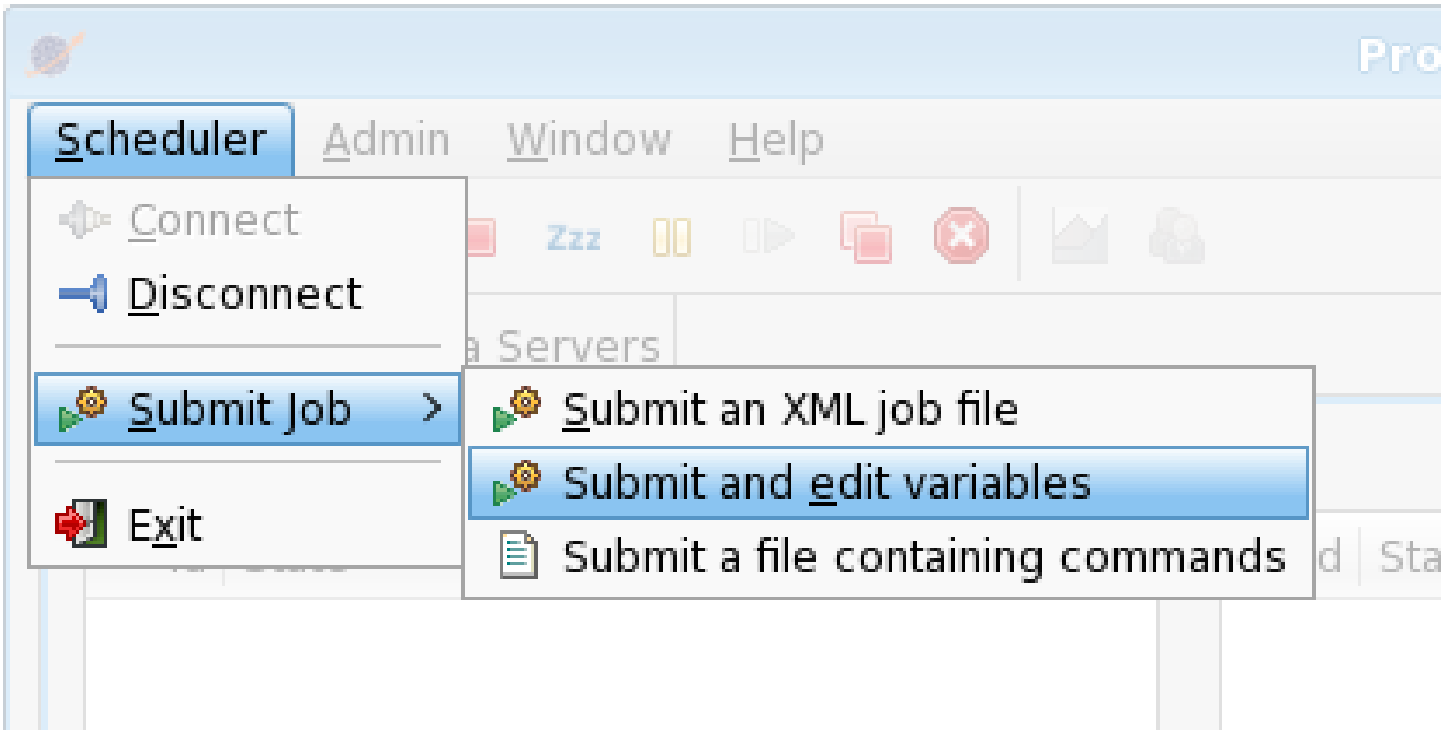


Figure 5.15. Submit a Job and edit variables

For each submitted job, all the variables defined in the **<variables>** tag can be edited. This enables users to submit jobs with different runtime parameters for everything that can be expressed in the descriptor with variables.

For each variable found in the descriptor, the dialog suggests the following values in a Combo list:

- The default value read in the descriptor
- The URL of each running Data Server defined in the Data Servers view. This allows specifying custom InputSpace or OutputSpace for the job, using the ones created in the client.

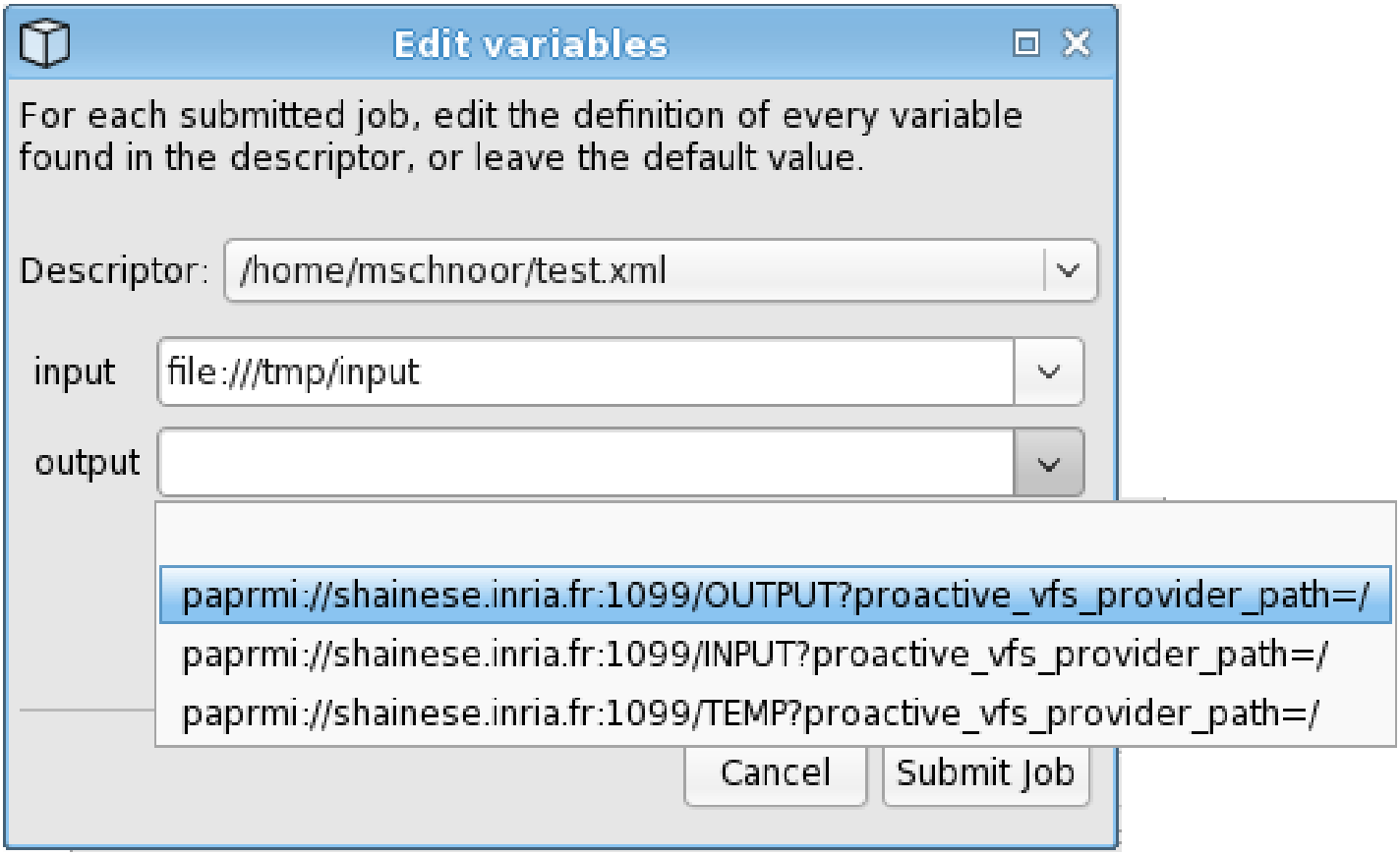


Figure 5.16. Variables edition

5.4.9. Command file job submission

Once you have defined a file containing the path to native commands to submit (see Section 2.1.5, “Create a job from a simple flat file”), this wizard allows you to submit it. Main page provides you with a way to specify additional parameters for your job:

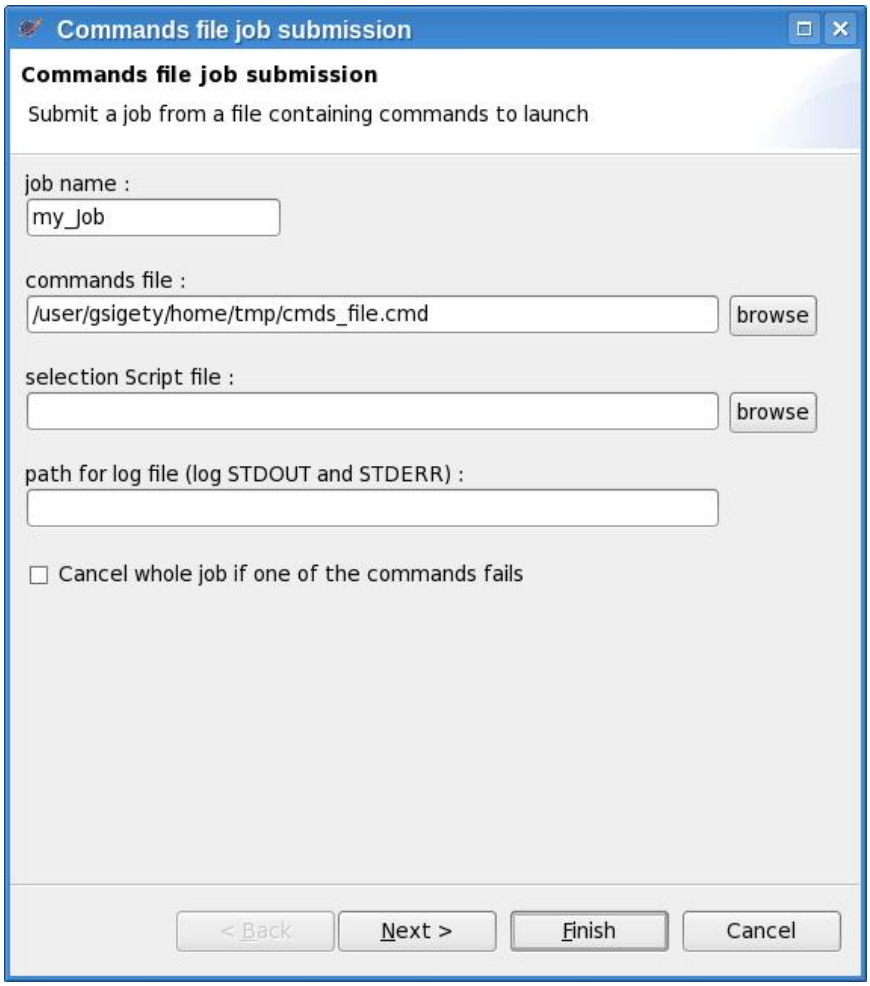


Figure 5.17. Control Panel

- **job name** - enter a name for your job.
- **command file** - click on browse button for choosing your file containing native commands using a file chooser dialog. This file chooser dialog shows only files suffixed by ".cmd".
- **selection script** - click on browse button and choose a selection script to associate to each native task of your job.
- **path for log file** - enter a path to a log file that will contain all STDOUT and STDERR of job execution. This file is created (emptied if exits) at each beginning of job execution.
- **Cancel whole job if one of the commands fails** - if you check that, the Scheduler will stop all executions of your commands if one fails. Scheduler assumes that a failed native command is a command that has a return code different from 0.

Clicking on the Next button, you will see a brief summary of your job:

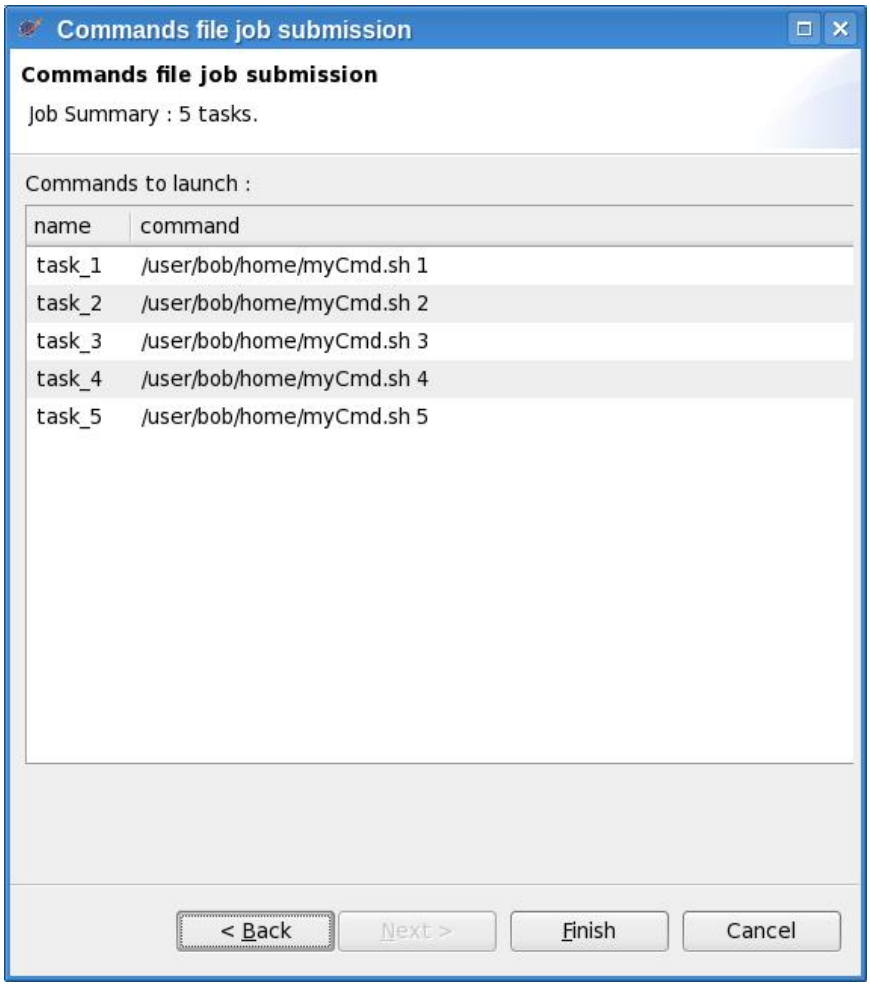


Figure 5.18. Control Panel

Click on Finish to launch submission. Native commands are scheduled in parallel. If there are enough free resources (nodes) for every command, all commands will be launched simultaneously.

5.4.10. Remote Connection

Sometimes, users want to provide visual feedback of the result of a task. It can then be necessary to connect on a remote host to observe specific data, or launch a specific application.

One specific scenario can be that a VNC server is started, and graphical clients should be able to connect once the task that creates the data to observe is terminated.

To achieve this, the **task** has to print in its standard output the following string: 'PA_REMOTE_CONNECTION;10005;vnc;localhost:5901'. All but the fields separated by semicolons, except the first, must be adapted:

- **PA_REMOTE_CONNECTION** : this is a mandatory constant to identify the String
- **10005** : this is the Task ID. It will allow the graphical client to bind this specific string to a task when reading the job output.
- **vnc** : this is a generic application type that will be used by the graphical client to associate a specific application on the local machine.
- **localhost:5901** : the URL that will be contacted.

On the client side, users need to define which application will be associated depending the type read. This can be controlled by adding entries in the Window -> Preferences -> Remote Connection dialog.

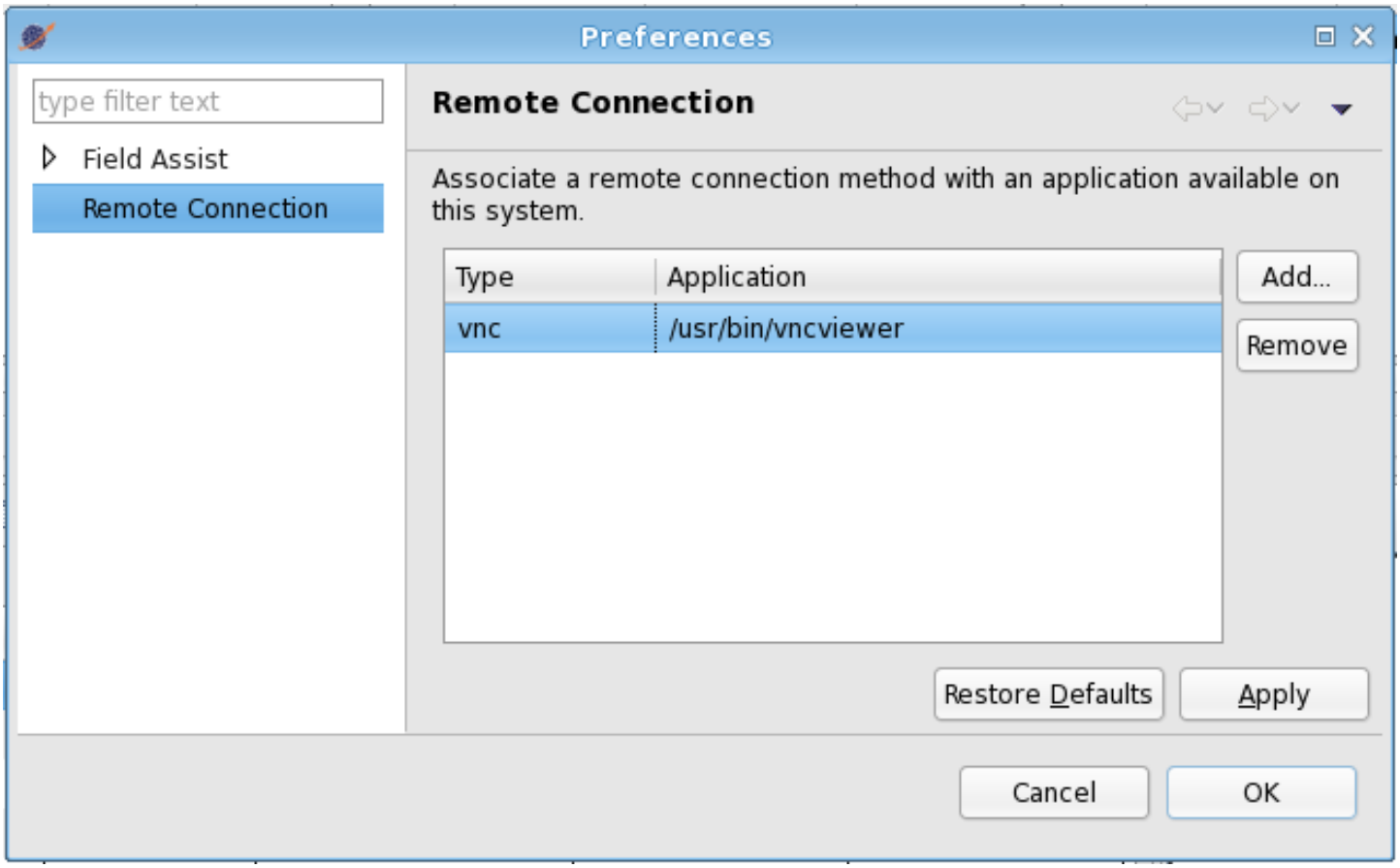


Figure 5.19. Application association preferences

Then, when the appropriate task terminates, simply select it in the Task View and click 'Remote Connection' in the context menu or icon bar. If the identification String was read and correctly formed, the remote connection application will be started.

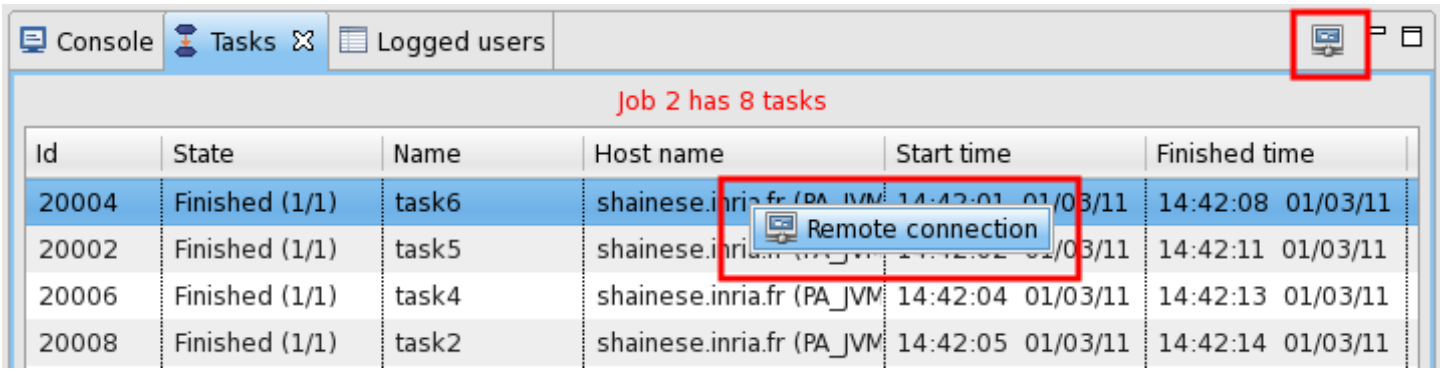


Figure 5.20. Application association preferences

Part II. ProActive Scheduler's Extensions

Table of Contents

Chapter 6. ProActive Scheduler's Matlab Extension

6.1. Presentation

6.1.1. Motivations

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. As of 2004, MATLAB was used by more than one million people in industry and academia.

The goal of this extension is to equip Matlab with a generic interface to Grid computing. This extension allows the deployment of Matlab instances on several nodes of the grid (and to use these instances like computing engines) and submitting of Matlab tasks over the grid. These engines are monitored by a central ProActive API. A natural condition is to deploy an application (based on this interface) strictly on hosts where the Matlab software is installed.

6.1.2. Features

The interface with Matlab inside ProActive comes as a Matlab Toolbox that can be directly accessed from the Matlab environment (ProActive Scheduler Toolbox). The objectives of **ProActive Scheduler Toolbox** are to provide you with tools that:

- Allow you to run Matlab functions on remote computers.
- Do not block the local Matlab session while remote results are being produced.
- Allow you to seamlessly retrieve results when you need them, just as if the functions were run locally.
- Provide you detailed remote log/output information, altogether with errors if any occurred.
- Allow a disconnected mode, jobs can be submitted and the matlab session doesn't need to remain active while the job is processing.
- Allow automatic source transfer, data file transfer, transfer of local matlab workspace and other configurable options.

6.2. Installation for Matlab

The toolbox can be found at **SCHEDULING/extensions/matlab/toolbox**. To install it inside Matlab, simply add the path to the toolbox to matlab search path, via the **addpath** command at the matlab prompt:

```
>> addpath('/home/fviale/eclipse_workspace/Scheduling_Clean/extensions/matlab/toolbox/')
```

The full documentation of **ProActive Scheduler Toolbox** will then be available on the Matlab Help:

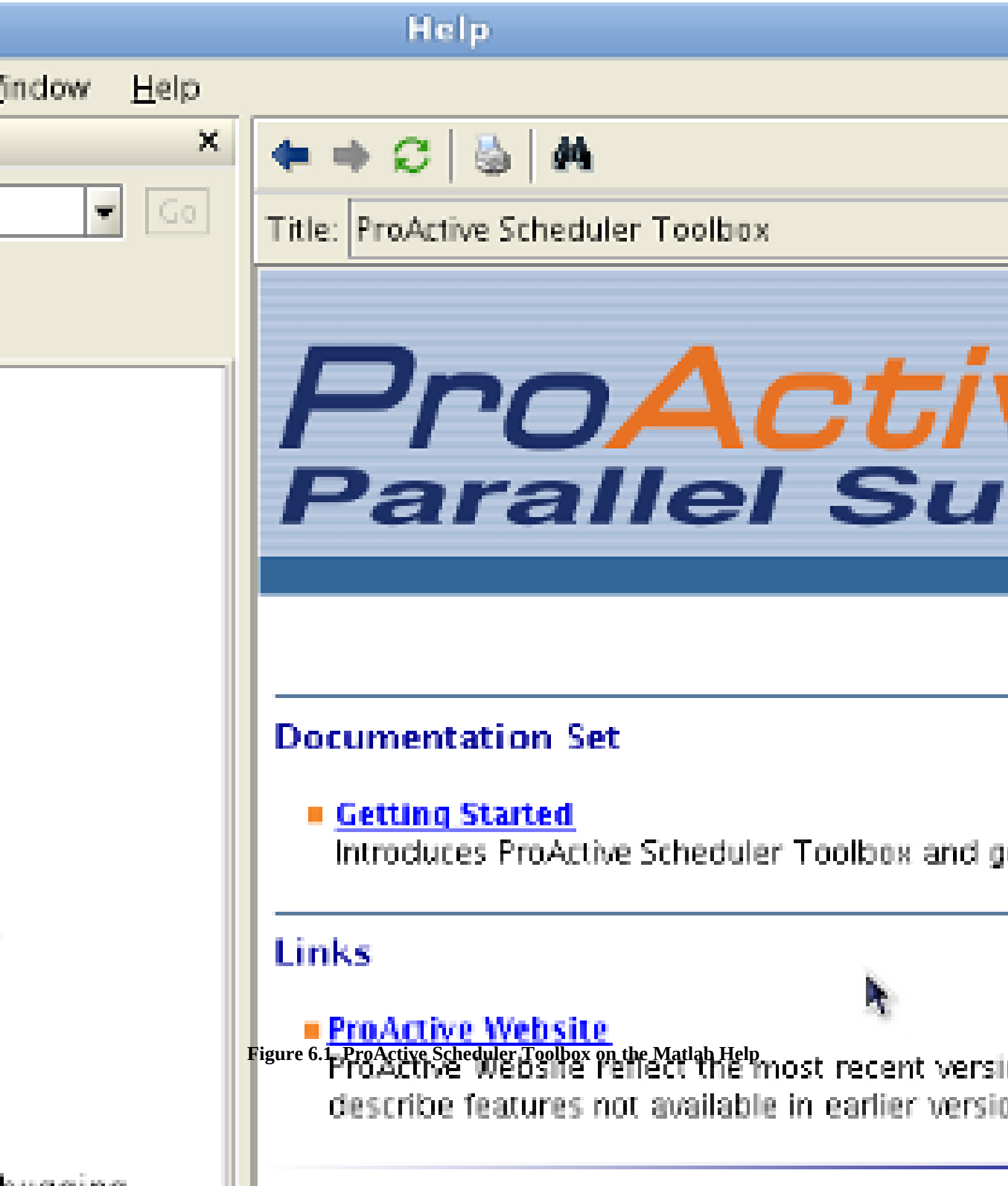


Figure 6.1. ProActive Scheduler Toolbox on the Matlab Help

Please refer to the Matlab Help for a detailed explanation of ProActive Scheduler Toolbox usage.

Alternatively you can download the pdf version of this help : **ProActiveSchedulerToolbox.pdf**[1] or browse it online **http://proactive.inria.fr/release-doc/Matlab/**

[1] http://proactive.inria.fr/release-doc/Matlab/ProActiveSchedulerToolbox.pdf

Chapter 7. ProActive Scheduler's Scilab Extension

7.1. 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).

The interface with Scilab inside ProActive comes as a Scilab Toolbox that can be directly accessed from the Scilab environment (ProActive Scheduler Toolbox). The objectives of **ProActive Scheduler Toolbox** are to provide you with tools that:

- Allow you to run Scilab functions on remote computers.
- Do not block the local Scilab session while remote results are being produced (being developped).
- Allow you to seamlessly retrieve results when you need them, just as if the functions were run locally.
- Provide you detailed remote log/output information, altogether with errors if any occurred.
- Allow automatic source transfer, data file transfer, transfer of local Scilab workspace and other configurable options.

7.2. Installation for Scilab

The toolbox is currently under Alpha version. Therefore we provide only help to install the toolbox but not of its usage. Next releases of ProActive Scheduler will feature a fully functional and documented toolbox.

The toolbox works only with Scilab version **5.3**. The toolbox can be found at **SCHEDULING/extensions/scilab/PAscheduler**. To install it inside Scilab, the toolbox must be built and loaded inside Scilab via the standard Scilab toolboxes installation scheme. The toolbox depends on another toolbox called **JIMS** which can be found at **SCHEDULING/extensions/scilab/JIMS**. Below is the sequence of instructions from the scilab prompt to install both toolboxes.

Installation of JIMS:

```
-->cd('/home/fviale/eclipse_workspace/Scheduling_Clean/extensions/scilab/JIMS')
 ans  =

 /home/fviale/eclipse_workspace/Scheduling_Clean/extensions/scilab/JIMS

-->exec builder_src.sce

-->mode(-1);
  Génère un fichier gateway
  Génère un fichier loader
  Génère un Makefile
  ilib_gen_Make : Copie les fichiers de compilation (Makefile*, libtool...) vers TMPDIR
  ilib_gen_Make : Copie unwrapshort.cpp vers TMPDIR
  ilib_gen_Make : Copie getmethods.cpp vers TMPDIR
  ilib_gen_Make : Copie unwrapstring.cpp vers TMPDIR
  ilib_gen_Make : Copie unwrapboolean.cpp vers TMPDIR
  ilib_gen_Make : Copie GiwsException.cpp vers TMPDIR
  ilib_gen_Make : Copie ScilabObjects.cpp vers TMPDIR
  ilib_gen_Make : Copie unwrapdouble.cpp vers TMPDIR
```

```
  ilib_gen_Make : Copie unwrapfloat.cpp vers TMPDIR
  ilib_gen_Make : Copie unwrapint.cpp vers TMPDIR
  ilib_gen_Make : Copie ScilabJavaClass.cpp vers TMPDIR
  ilib_gen_Make : Copie unwraplong.cpp vers TMPDIR
  ilib_gen_Make : Copie unwrapchar.cpp vers TMPDIR
  ilib_gen_Make : Copie ScilabJavaObject.cpp vers TMPDIR
  ilib_gen_Make : Copie unwrapbyte.cpp vers TMPDIR
  ilib_gen_Make : Copie ScilabJavaArray.cpp vers TMPDIR
  ilib_gen_Make : Copie ScilabJavaObject2.cpp vers TMPDIR
  ilib_gen_Make : Copie ScilabClassLoader.cpp vers TMPDIR
  ilib_gen_Make : Copie getfields.cpp vers TMPDIR
  ilib_gen_Make : Copie ScilabObjectsk.c vers TMPDIR
  ilib_gen_Make : Copie getSciArgs.c vers TMPDIR
  ilib_gen_Make : Copie sci_convMatrixMethod.c vers TMPDIR
  ilib_gen_Make : Copie sci_displayJObj.c vers TMPDIR
  ilib_gen_Make : Copie sci_getFields.c vers TMPDIR
  ilib_gen_Make : Copie sci_getMethods.c vers TMPDIR
  ilib_gen_Make : Copie sci_getRep.c vers TMPDIR
  ilib_gen_Make : Copie sci_getfield.c vers TMPDIR
  ilib_gen_Make : Copie sci_getfield_l.c vers TMPDIR
  ilib_gen_Make : Copie sci_getfield_lu.c vers TMPDIR
  ilib_gen_Make : Copie sci_getfield_u.c vers TMPDIR
  ilib_gen_Make : Copie sci_import.c vers TMPDIR
  ilib_gen_Make : Copie sci_import_u.c vers TMPDIR
  ilib_gen_Make : Copie sci_invoke.c vers TMPDIR
  ilib_gen_Make : Copie sci_invoke_l.c vers TMPDIR
  ilib_gen_Make : Copie sci_invoke_lu.c vers TMPDIR
  ilib_gen_Make : Copie sci_invoke_u.c vers TMPDIR
  ilib_gen_Make : Copie sci_javaArray.c vers TMPDIR
  ilib_gen_Make : Copie sci_javaCast.c vers TMPDIR
  ilib_gen_Make : Copie sci_javadeff.c vers TMPDIR
  ilib_gen_Make : Copie sci_loadClass.c vers TMPDIR
  ilib_gen_Make : Copie sci_newInstance.c vers TMPDIR
  ilib_gen_Make : Copie sci_newInstance_l.c vers TMPDIR
  ilib_gen_Make : Copie sci_remove.c vers TMPDIR
  ilib_gen_Make : Copie sci_setfield.c vers TMPDIR
  ilib_gen_Make : Copie sci_setfield_l.c vers TMPDIR
  ilib_gen_Make : Copie sci_unwrap.c vers TMPDIR
  ilib_gen_Make : Copie sci_wrap.c vers TMPDIR
  ilib_gen_Make : Copie sci_wrapinchar.c vers TMPDIR
  ilib_gen_Make : Copie sci_wrapinfloat.c vers TMPDIR
  ilib_gen_Make : Copie sci_wrapvar.c vers TMPDIR
  ilib_gen_Make : Copie libimportjava.c vers TMPDIR
  ilib_gen_Make : configure : Génère le Makefile.
  ilib_gen_Make : Modification du Makefile dans TMPDIR.
  Exécute le makefile
  Génère un fichier cleaner

-->exec start.sce

-->here=get_absolute_file_path('start.sce');

-->macros=here+'macros';
```

```
-->java_home='/home/fviale/bin/jdk1.6.0_64/';

-->exec loader.sce;
Bibliothèque partagée chargée.
Edition de liens effectuée.

-->impjava=lib(macros);

-->if isfile('classpath/jlatexmath-0.9.3.jar') then
-->    javaclasspath([here+'jar/ScilabObjects.jar' here+'classpath' 'classpath/jlatexmath-0.9.3.jar' here+'classpath/
scilatex.jar']);
-->else
-->    disp('Warning, cannot use latex library, in order to use it, download it at http://forge.scilab.org/index.php/p/
jlatexmath/downloads/130/ and put the jar file into the classpath subfolder');

 Warning, cannot use latex library, in order to use it, download it at http://forge.scilab.org/index.php/p/jlatexmath/
downloads/130/ and put the jar file into the c
      lasspath subfolder
-->    javaclasspath([here+'jar/ScilabObjects.jar']);
-->end
```

Installation of PAScheduler:

```
-->cd ../PAscheduler/
 ans  =

 /home/fviale/eclipse_workspace/Scheduling_Clean/extensions/scilab/PAscheduler

-->exec builder.sce

-->mode(-1);

-->exec loader.sce

-->// ============================================================================

-->// generated by builder.sce

-->// Copyright INRIA 2008

-->// ============================================================================

-->try
--> getversion('scilab');
-->catch
--> warning('Scilab 5.0 or more is required.');
--> return;
-->end;

-->// ============================================================================

-->root_tlbx = get_absolute_file_path('loader.sce');

-->exec(root_tlbx+'etc\'+'toolbox_pascheduler.start');
```

```
Start Toolbox PAScheduler

Load macros

Load gateways

Load help

-->// ===========================================================================

-->clear root_tlbx;

-->// ===========================================================================
```

If those steps were done successfully the toolbox will be available to use on Scilab

Chapter 8. Scheduling examples of Modelica simulations

8.1. Presentation

8.1.1. Motivations

Modelica is an object oriented, equation based language, used to model complex physical systems.

Similarly to ProActive Scheduler's Matlab and Scilab extensions, the Modelica extension comes in the following ways:

- The ability to write simple Modelica tasks workflows and submit these workflows to the ProActive Scheduler (with the possibility to define dependencies between tasks and transfering results from one task to the other).
- A parallelization facility to parallelize scripts and retrieve results. Monitoring of these tasks will still be possible through the Scheduler interface.

8.2. Installation

Before trying to run modelica simulations, you need to install Modelica in your environment.

Follow the instructions for the installation of the **Open Modelica Compiler** http://www.ida.liu.se/labs/pelab/modelica/OpenModelica.html#Download

Set the environment variable **SENDDATALIBS** with the following value:

-lsendData -lQtNetwork-mingw -lQtCore-mingw -lQtGui-mingw -luuid -lole32 -lws2_32

8.3. Modelica Scheduler Jobs

We'll write a simple Modelica job example. . This example will solve numerically the Bouncing Ball problem.

```
model BouncingBall
  parameter Real e = 0.9 "coefficient of restitution";
  parameter Real g = 9.81 "gravity acceleration";
  Real h(start = 1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start = true) "true, if ball is flying";
  Boolean impact;
  Real v_new;
  Integer foo;

equation
  impact = h <= 0.0;
  foo = if impact then 1 else 2;
  der(v) = if flying then  -g else 0;
  der(h) = v;
  when {h <= 0.0 and v <= 0.0,impact} then
     v_new = if edge(impact) then  -e * pre(v) else 0;
   flying = v_new > 0;
   reinit(v, v_new);

  end when;
```

```
end BouncingBall;
```

8.3.1. Tasks split

The omc compiler is used in several steps, to make the source code, build and run the executable.

Generate the **BouncingBall.makefile**

```
<taskFlow>
  <task name="omc">
    <description>generates makefile file</description>
    <selection>
      <!-- This job descriptor works only on windows -->
      <script type="static">
        <file path="${pa.scheduler.home}/samples/scripts/selection/checkWindows.js"/>
      </script>
    </selection>
    <nativeExecutable>
      <staticCommand value="${MODELICA_BIN_DIR}/omc" workingDir="${WORK_DIR}">
        <arguments>
          <argument value=" +s "></argument>
          <argument value="${WORK_DIR}\BouncingBall.mo"></argument>
        </arguments>
      </staticCommand>
    </nativeExecutable>
  </task>
```

Compile the **BouncingBall.makefile** to generate the executable **BouncingBall.exe**

Note that the argument containing the modelname (BouncingBall) is passed without extension

```
<task name="compile">
  <description> compiles makefile </description>
  <depends>
    <task ref="omc"/>
  </depends>
  <nativeExecutable>
    <staticCommand value="${MODELICA_BIN_DIR}\Compile.bat" workingDir="${WORK_DIR}">
      <arguments>
        <argument value="BouncingBall"></argument>
      </arguments>
    </staticCommand>
  </nativeExecutable>
</task>
```

Executing can be achieved directly by adding another task to the job

```
 <task name="run">
  <description> executes </description>
   <depends>
     <task ref="compile"/>
   </depends>
  <nativeExecutable>
    <staticCommand value="${WORK_DIR}\BouncingBall.exe" workingDir="${WORK_DIR}"></staticCommand>
```

```
  </nativeExecutable>
</task>
```

or invoking the executable directly from the command line.

8.3.2. MOS file

Alternatively, one can create a mos-file that performs the above mentioned tasks

```
loadModel(Modelica);
loadFile("BouncingBall.mo");
simulate(BouncingBall,stopTime = 2);
```

In this way, one is able to set the simulation parameters (e.g. stopTime) externally, otherwise the default would be startTime = 0, stopTime = 1.

The xml job is built in a similar way as above. Now there will be only one task performing all the operations.

```
<taskFlow>
  <task name="task1">
    <description>runs the simulation</description>
    <selection>
      <!-- This job descriptor works only on windows -->
      <script type="static">
        <file path="${pa.scheduler.home}/samples/scripts/selection/checkWindows.js"/>
      </script>
    </selection>
    <nativeExecutable>
      <staticCommand value="${MODELICA_BIN_DIR}/omc" workingDir="${WORK_DIR}">
        <arguments>
          <argument value=" +s "></argument>
          <argument value="${WORK_DIR}/BouncingBall.mos"></argument>
        </arguments>
      </staticCommand>
    </nativeExecutable>
  </task>
```

8.4. Handling results

The simulation results will be output in a file named **BouncingBall_res.plt**. Results (all the variables) will be plot by invoking the following command

```
>> doPlot BouncingBall_res.plt
```

Chapter 9. ProActive Scheduler's Files Split-Merge Extension

9.1. Presentation

The **"Files Split Merge"** generic framework provides an easy way to develop a distribution layer for a native application in order to run it on a ProActive managed infrastructure through the ProActive Scheduler and Resource Manager. Here is a short description of this framework: A distribution layer can be implemented, with this framework, for native applications that meet these requirements:

- the input data can be split into slices so that an instance of the application can be run against each slice of data
- there is a way to merge the outputs of all runs (each output corresponding to one slice of the input data) in order to obtain the same final output as the one generated by the application using the entire input data

The framework implements the “split input data | execute multiple instances | merge results” pipeline, so that applications built over this framework provide features like:

- **disconnected mode**: the user can disconnect from the application (close the application) during the execution of his "run" by the ProActive Scheduler Server. Once the application restarted, it will ask the server for the results and perform the merging operation
- **fault tolerance**: the entire system is fault tolerant - the application itself and the ProActive Scheduler and Resource Manager servers. The granularity of the fault tolerance is a task (an instance of the native application running against a slice of data) which means that, if a failure occurs into the system, only the tasks affected by this failure will be rerun.
- **keep track of the advancement of the "run"** and notify the user when the results are available
- **manage the pushing of the input slices** to the remote compute resources and **the pulling of the results** (several copy protocols are available)
- **a textual user interface** for the distribution layer application
- **a graphical user interface**, provided as plugin for the Scheduler User Interface

The framework provides an API which allows the developer of the distribution layer application to specify the split and merge operations, define specific post treatment of data results, define the scripts that will be used for data copy as well as their arguments, specify the generation of the command which launches the native application, depending on the type of the remote computing resource (i.e. the remote operating system).

9.2. How-To – what you need to implement to make it work

Using the framework is straight-forward. Several abstract classes are to be extended in order to define application-specific implementation. A quick overview is presented below.

Implement MyJobCreator class by extending JobCreator class

Initiate your Jobcreator: provide an init() method or a constructor which takes in argument the data you need for creating the job (i.e. input files, input parameters, etc.)

Implement the split data operation:

```
protected abstract List<File> splitData() throws IOException;
```

This method is responsible for splitting the input data for a job in n "slices", one for each task. The size of the returned list will determine the number of tasks that will be created. This list could contain, for instance, one input file for each task, or one folder per task, containing a set of input files for that task. Note that, each element of this list will be sent as argument, during the task creation process, to these methods (see details bellow):

```
protected List<String> getGenerationScriptArgs(int taskNb, File taskFileOnServer) {
   return new LinkedList<String>();
}


protected abstract List<String> getPreScriptArgs(int taskNb, File taskInputFileOnServer);


protected abstract List<String> getPostScriptArgs(int taskNb, File taskFileOnServer);


protected abstract List<String> getCleanScriptArgs(int taskNb, File taskFileOnServer);


protected String createCommandLineForTask(File taskFileOnServer, int taskNb) {
   return "";
}
```

Define what command should be executed on the remote machines. You can implement this method

```
protected String createCommandLineForTask(File taskFileOnServer, int taskNb) {
   return "";
}
```

and provide the command line to be executed as return of this method. If the command line to be executed depends on the remote operating system (or some remote parameters), you can override the methods

```
protected String getGenerationScriptFilePath() {
   return null;
}
```

and

```
protected List<String> getGenerationScriptArgs(int taskNb, File taskFileOnServer) {
   return new LinkedList<String>();
}
```

in order to provide the file path for the script that will generate the command and its arguments.

Define the file transfer in your application. The file transfer is to be performed by pre and post scripts. These scripts run on the remote nodes, before and respectively after the execution of the task itself. The pre script will usually be used for copying files from their initial location to the node and the post script to copy results from the node to a result folder accessible by the application. Several file transfer helpers (java implementation for several file transfer protocols) are provided in the Scheduler release 1.0, as well as sample java scripts for file transfer. You must implement the methods

```
protected abstract String getPreScriptFilePath();
```

and

```
protected abstract String getPostScriptFilePath();
```

in order to specify the files containing your pre and post scripts, as well as the methods

```
protected abstract List<String> getPreScriptArgs(int taskNb, File taskInputFileOnServer);
```

```
protected abstract List<String> getPostScriptArgs(int taskNb, File taskFileOnServer);
```

in order to provide the input arguments for the scripts. These methods will be called when the tasks are created.

Provide a script and arguments for cleaning the remote resources after use, by implementing the methods

```
protected abstract String getCleanScriptFilePath();
```

and

```
protected abstract List<String> getCleanScriptArgs(int taskNb, File taskFileOnServer);
```

Implement the JobConfiguration interface

A JobConfiguration object will be attached to a job when submitted to the Scheduler. This object should contain all the information needed in order to merge the results in the post- treatment process. This object should be implemented as a bean, with all attributes of type String: provide getters and setters methods for all attributes, by respecting the template

```
public String getAttribute();
public void   setAttribute(String val);
```

The JobConfiguration object will be passed as argument for the mergeResults(..) method in the PostTreatmentManager. The JobConfiguration object should provide all the information necessary to the merging of results.

Implement MyPostTreatmentManager class by extending JobPostTreatmentManager

Implement the method:

```
protected abstract void mergeResults(JobConfiguration jc, int numberOfTasks);
```

The JobConfiguration object should contain all the information needed to merge the results (result file names and locations, parameters, output file names, etc.)

(Optional) Write a ResultPreview class

Extend org.ow2.proactive.scheduler.common.task.ResultPreview in order to define how the partial results will be displayed in the Scheduler User Interface.

(Optional) Write a MenuCreator class

Extend GeneralMenuCreator class in order to define a menu for the Textual User Interface (if you need one). Also write command classes (extend Command) and add them to the MenuCreator (i.e. SubmitJobCommand). Once you have implemented the classes described above, you have to **initialize your application** by calling:

```
EmbarrasinglyParrallelApplication.instance().initApplication(url, userName, passwd, GoldMenuCreator.class,
   GoldJobPostTreatmentManager.class, GoldJobConfiguration.class);
```

If you don't need a textual user interface, just pass null for the MenuCreatorClass argument. In order to submit a job (from your main class or from a command in the textual user interface) you will only have to create a MyJobConfiguration object, initiate a MyJobCreator object and call the MyJobCreator#submitJob() method.

9.3. The framework functioning – what you don't need to implement since it's already there

When the Application is initiated (EmbarrassinglyParallelApplication.#initApplication(..) method is called) these steps are performed:

- an application Logger is created.
- the Textual UI is created (if needed) and attached as an appender to the Logger, so the user can see logged messages on the textual UI.
- a proxy to the scheduler (SchedulerProxy) is created, which is an active object which keeps a connection to the Scheduler and plays the Scheduler role for the local application.
- the ProstTreatmentManager object is initiated. It reads a file on system to find out if there are any jobs submitted in previous sessions that need post treatment operations. If there are any, it will start the post treatment process in an asynchronous way.
- a SchedulerEventListener is created which is an active object that will listen for events from the scheduler. It will notify the PostTreatmentManager and the user interface when results are available on the Scheduler.
- the user interface (if any) is started.

When a job is submitted by calling MyJobCreator.submitJob(..), these steps are performed:

- a TaskFlowJob is created
- the splitData() method is called – the return is a list of File objects, of size n
- n native tasks are created. During the creation of each task, these methods are called: getPreScriptFilePath(), getPreScriptArgs(..), getPostScriptFilePath(), getPostScriptArgs(..), getCleanScriptFilePath(), getCleanScriptArgs(..), createCommandLineForTask(..), getGenerationScriptFilePath(), getGenerationScriptArgs(...).
- the JobConfiguration object is attached to the job as generic information.
- the job is submitted to the Scheduler (through the local Scheduler Proxy)
- the Post Treatment Manager is informed that results are awaited for this job
- the Post Treatment Manager will save its new state to a file so that it can recover in case of shut-down or failure.

When a job is finished these steps are performed:

- the Post Treatment Manager will receive an event from the Scheduler Local Listener
- if the job succeed the Post Treatment Manager will get the results from the Scheduler, recreate the JobConfiguration object (that has been attached to the job) and merge the results in an asynchronous way.

Chapter 10. ProActive Scheduler's MapReduce Extension

10.1. Introduction to MapReduce

MapReduce is a kind of parallel programming model that derives from the map and reduce combinators present in functional languages like Lisp. In Lisp, the map takes as input a function and a sequence of values. It then applies the function to each value in the sequence. The reduce combines all the elements of a sequence using a binary operation. For example, it can use a function that implements the sum function to add up all the elements in the sequence. MapReduce is inspired by these concepts.

MapReduce was firstly introduced by Google whose aim for process large amounts of raw data, for example,crawled documents or web request logs using clusters of commodity hardware. The data to elaborate are so large, they must be distributed across thousands of machines in order to be processed in a reasonable time. The distribution implies parallel computing since the same computations are performed on each CPU, but with a different dataset.

MapReduce is an abstraction that allows Google engineers to perform simple computations while hiding the details of parallelization, data distribution, load balancing and fault tolerance. In fact, MapReduce programs are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of splitting the input data, scheduling the program's execution across the set of machines, handling machines failures and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system. The only requirements are: the programmer assure that the application he wants parallelize is data parallel and he must define two functions: **map** and **reduce**. Each function has key-value pairs as input and output. The type of the key and/or value can be defined by the user. Input data can be in any format as far as they can be loaded into key-value pairs by a user defined function. The map function takes the input key-value pairs and produces a bag of intermediate key-value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the reduce function. The reduce function, also written by the programmer, accepts an intermediate key and the bag of values for that key and it merges together the values received for that key.

The Figure 10.1, "Hadoop MapReduce execution" shows the execution of the Google MapReduce application.

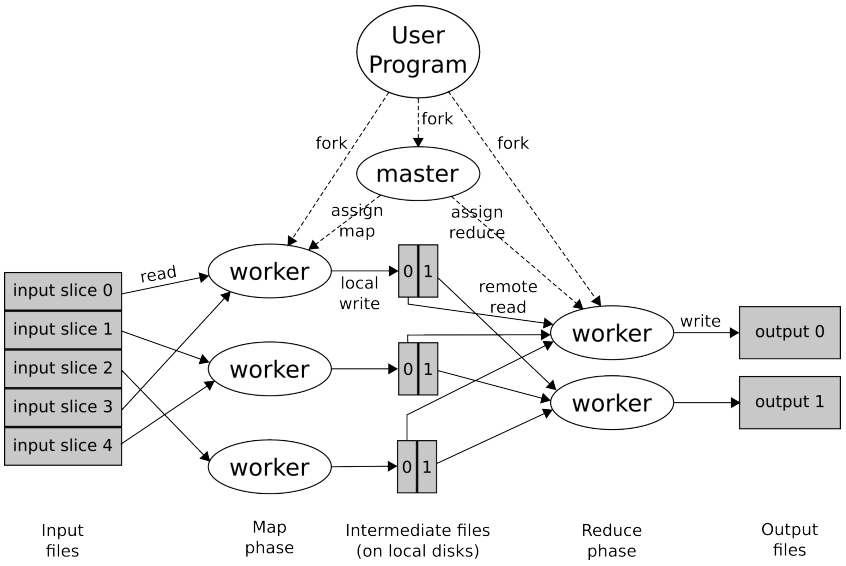


Figure 10.1. Hadoop MapReduce execution

The user must notice that the reduce phase can begin only after all the map tasks are done. However, this is the only needed synchronization point and the only inter-process communication.

The most important features of the MapReduce parallel programming model are: fault tolerance and data locality. The target architecture of the Google MapReduce is a cluster of thousand of commodity machines (e.g., 2-4 GB of main memory while networking hardware bandwith is 100 megabits/second). This means machine failures are common, re-execution of map and/or reduce tasks is the primary mechanism for fault tolerance. The data to elaborate that are stored on the disks are managed by a distributed file system, GFS (Google File System), that uses replication to provide availability and reliability on top of unreliable hardware. Moreover, the bandwidth is conserved because GFS file blocks are stored on the local disks of the machines that make up the Google cluster. The elaboration is data local because each user defined map function elaborates the data replica stored on the same machine it executes on (or the map function elaborates the data replica stored in the nearer machines). This means we save the bandwidth.

Google MapReduce is a C++ library, but some Java and open source implementation also exist: Apache Hadoop MapReduce. Hadoop MapReduce follows the same approach taken by Google, so that the system takes in charge the communication between the machines. The user must only implement the **map** and **reduce** functions and run the job specifying the input data and the output directory where he expects to find results. Moreover, Hadoop uses a distributed file system, HDFS (Hadoop Distributed File System), like Google.

The architecture and the execution of Hadoop MapReduce job is presented in the Figure 10.2, “Hadoop MapReduce execution”.

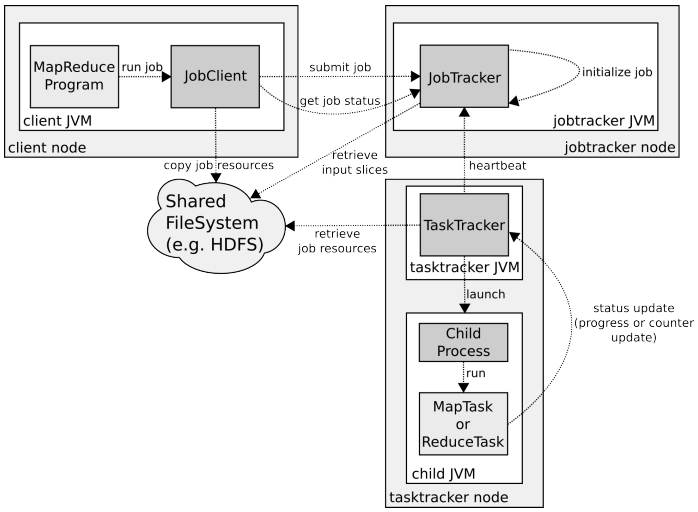


Figure 10.2. Hadoop MapReduce execution

At the highest level there are four independent entities:

- The **client** which submits the MapReduce job;
- The **jobTracker**, a java applications running as a background process on each Hadoop cluster node, which coordinates the job execution;
- The **taskTracker**, a java applications running as a background process on each Hadoop cluster node, which runs the tasks (map and/or reduce) the job is split into;
- The shared file system, usually **HDFS** (Hadoop Distributed File System) which is used for sharing job files between jobtracker and taskTracker entities.

The architecture shown in the Figure 10.2, “Hadoop MapReduce execution” is quite similar to the ProActive Scheduler one (the job is submitted to the ProActive Scheduler, the ProActive Scheduler and the ProActive Resource Manager coordinate the ProActive job execution, the ProActive node runs the tasks belonging the ProActive job) even if in ProActive there is not a distributed file system. We implemented the ProActive MapReduce HAdoop-like API that allows the user to execute the Hadoop MapReduce job using the ProActive Scheduler - Resource Manager architecture.

10.2. ProActive MapReduce Hadoop-like API

The **Hadoop-like** definition of the ProActive MapReduce API is due to the fact that the user can define the MapReduce job as when he uses Hadoop MapReduce. The only difference is that to deal with the differences between ProActive and Hadoop MapReduce, the

Hadoop-like API we defined requires additional configuration informations from the users when building the ProActive MapReduce job. Moreover, considering that "$SCHEDULER_HOME" represents the home (or root) folder of the ProActive Scheduler used, the user must add the following jars to the classpath of the ProACtive MapReduce job.

- $SCHEDULER_HOME/lib/bouncycastle.jar
- $SCHEDULER_HOME/lib/commons-codec-1.3.jar
- $SCHEDULER_HOME/lib/commons-logging-1.1.1.jar
- $SCHEDULER_HOME/lib/hadoop/hadoop-0.20.2-core.jar
- $SCHEDULER_HOME/lib/http-2.0.4.jar
- $SCHEDULER_HOME/lib/javassist.jar
- $SCHEDULER_HOME/lib/common/script/js.jar
- $SCHEDULER_HOME/lib/log4j.jar
- $SCHEDULER_HOME/lib/netty-3.2.0.ALPHA2.jar
- $SCHEDULER_HOME/dist/lib/ProActive_ResourceManager.jar
- $SCHEDULER_HOME/dist/lib/ProActive_ResourceManager-client.jar
- $SCHEDULER_HOME/dist/lib/ProActive_Scheduler-client.jar
- $SCHEDULER_HOME/dist/lib/ProActive_Scheduler-core.jar
- $SCHEDULER_HOME/dist/lib/ProActive_Scheduler-fsm.jar
- $SCHEDULER_HOME/dist/lib/ProActive_Scheduler-mapreduce.jar
- $SCHEDULER_HOME/dist/lib/ProActive_Scheduler-worker.jar
- $SCHEDULER_HOME/dist/lib/ProActive_SRM-common.jar
- $SCHEDULER_HOME/dist/lib/ProActive_SRM-common-client.jar
- $SCHEDULER_HOME/dist/lib/ProActive_utils.jar
- $SCHEDULER_HOME/dist/lib/ProActive.jar
- $SCHEDULER_HOME/lib/script-js.jar

To build the ProActive MapReduce job the user must specify the Hadoop MapReduce job and the additional configuration: **org.ow2.proactive.scheduler.ext.mapreduce.PAMapReduceJobConfiguration**. In this way the ProActive MapReduce job can be build as follow:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapreduce.Job;
import org.ow2.proactive.scheduler.ext.mapreduce.PAMapReduceJob;
import org.ow2.proactive.scheduler.ext.mapreduce.PAMapReduceJobConfiguration;

...

Job job = new Job( new Configuration(), "WordCount");
job.setMapperClass( ... );
...

FileInputFormat.addInputPath( job, new Path( "path/to/input/fileOrFolder" ) );
FileOutputFormat.setOutputPath(job, new Path( "path/to/output/folder" ));

...

PAMapReduceJobConfiguration pamrjc = null;
File paMapReduceConfigurationFile = new File( "path/to/configuration/file" );
pamrjc = new PAMapReduceJobConfiguration( paMapReduceConfigurationFile );
```

```
// We build the ProActive MapReduce job
PAMapReduceJob pamrj = new PAMapReduceJob( hadoopJob, pamrjc );

// We run the ProActive MapReduce Job
if ( pamrj != null ) {
   if ( pamrj.run() ) {
      System.out.println( "The ProActive MapReduce job is correctly submitted!" );
   } else {
      System.out.println( "The ProActive MapReduce job is NOT submitted." );
   }
}
```

In the previous code statement the PAMapReduceJobConfiguration instance is created from a configuration file (actually a property file is required). Then the ProActive MapReduce job (the instance of the PAMapReduceJob class) is created using the Hadoop MapReduce job and the additional configuration PAMapReduceJobConfiguration as parameters.

The ProActive MapReduce Hadoop-like API, when the user create a MapReduce job and before it is submitted to the ProActive Scheduler, internally build a workflow made up of five tasks: SplitterPATask, MapperPATask, MapperJoinPATask, ReducerPATask and ReducerJoinPATask. The execution order of those tasks and their behavior is the following: the **SplitterPATask** creates the input splits from the input file. Each **MapperPATask** elaborate one and only one input split. This means that a replication occurs for the MapperPATask and that the replication degree is equal to the number of created input splits. The **MapperJoinPATask** implement the join of the execution of the MapperPATask replicas. This means that the **ReducerPATask** starts its execution only after the MapperJoinPATask (and so all the MapperPATask replicas) end. The ReducerPATask is replicated too. The number of replicas of the ReducerPATask is defined by the user. If the user does not define it, only one ReducerPATask is executed. The last task to execute is the **ReducerJoinPATask** that implements the join of the ReducerPATask replicas.

The Figure 10.3, “ProActive MapReduce workflow” shows the structure of the ProActive MapReduce workflow.

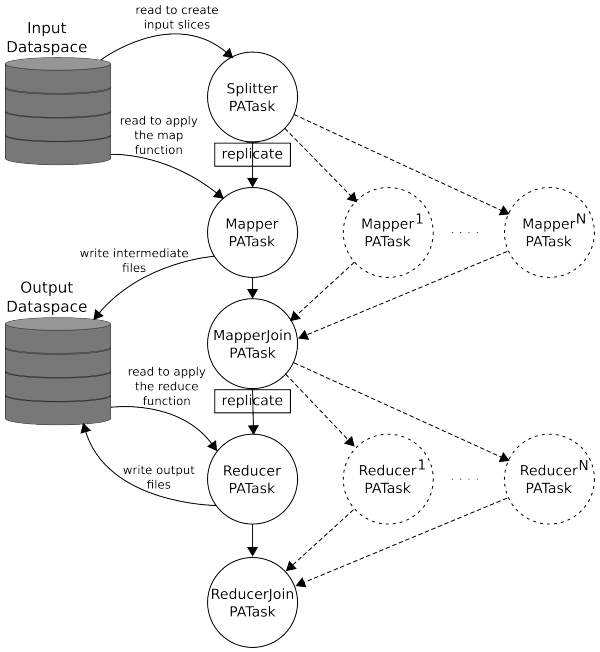


Figure 10.3. ProActive MapReduce workflow

The ProActive MapReduce job is actually a ProActive workflow. That means the additional configuration parameters the user must specify are directly related to the ProActive workflow and to the tasks belonging to it (e.g., the "cancelJobOnError" attribute of a task, the

input files of a task etc...). However, there are also some other parameters that the user must specify to overcome the existing differences between the Hadoop and ProActive MapReduce Hadoop-like APIs.

The user must notice that the ProActive MapReduce Hadoop-like API defines default values for many of the parameters he can specify (e.g., most of the ProActive task attributes have the corresponding default value), while the user is forced to define some other parameters (e.g., above all the input space and the output space of the ProActive MapReduce workflow).

The ProActive MapReduce Hadoop-like API requires that the user specify the following parameters:

- **org.ow2.proactive.scheduler.ext.mapreduce.schedulerUrl**: the URL to use to access to the ProActive Scheduler;
- **org.ow2.proactive.scheduler.ext.mapreduce.username**: the username to use to establish the connection to the ProActive Scheduler;
- **org.ow2.proactive.scheduler.ext.mapreduce.password**: the password to use to establish the connection to the ProActive Scheduler;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.inputSpace**: the input space of the job. The user must notice that the input files are in the root of that specified input space or in a sub-folder of it.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.outputSpace**: the output space of the job. the user must notice that the output files will be stored in a sub-folder of the output space and that they cannot be stored in the root of the output space.

All the other configuration parameters are optional as the ProActive MapReduce Hadoop-like API provides default values for them. Those optional parameter are the followings:

- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.cancelJobOnError**: the "cancelJobOnError" attribute of the ProActive MapReduce workflow;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.classpath**: the classpath of the ProActive MapReduce workflow. This classpath contains the user implementations of the Hadoop Mapper, Reducer, InputFormat etc...;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.description**: the "description" attribute of the ProActive MapReduce workflow;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.logFilePath**: the path to the file into which the log messages produced during the execution of the ProActive MapReduce workflow must be stored;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.maxNumberOfExecutions**: the "maxNumberOfExecutions" attribute of the ProActive MapReduce workflow;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.projectName**: the "projectName" attribute of the ProActive MapReduce workflow;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.restartTaskOnError**: the "restartTaskOnError" attribute of the ProActive MapReduce workflow;

The tasks specific attributes are optional too:

- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.[splitter|mapper|mapperJoin|reducer|reducerJoin]PATask.cancelJobOnError**: the "cancelJobOnError" attribute specific to the corresponding task;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.[splitter|mapper|mapperJoin|reducer|reducerJoin]PATask.description**: the "description" attribute specific to the corresponding task;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.[splitter|mapper|mapperJoin|reducer|reducerJoin]PATask.maxNumberOfExecutions**: the "maxNumberOfExecutions" attribute specific to the corresponding task;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.[splitter|mapper|mapperJoin|reducer|reducerJoin]PATask.name**: the "name" attribute specific to the corresponding task;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.[splitter|mapper|mapperJoin|reducer|reducerJoin]PATask.restartTaskOnError**: the "restartTaskOnError" attribute specific to the corresponding task.

Some other attributes are also optional but they are very important:

- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.splitterPATask.inputSplitSize**: defines the size, in bytes, of the input split (fragment of the input data) each mapper must elaborate. This configuration parameter is optional but no default value is

defined by the ProActive MapReduce Hadoop-like API. When the user does not specify the size of the input split then the default value is given by the user implemented (or used) Hadoop InputFormat. This means that the ProActive MapReduce Hadoop-like API uses the user specified Hadoop InputFormat class to create input splits and that the size of those input split is equal to a default value computed by the user specified Hadoop InputFormat class. We must also notice that when the user defines a value of 0 (zero) bytes for the size of the input split, then input splits with a size equal to the minimum possible value forecast by the Hadoop InputFormat class used are created. While when the user defines a size greater than the size of the input file then only one input split is created and the size of that input split is equal to the size of the input file.

- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.splitterPATask.readMode**: defines the read mode of the SplitterPATask. The two possible value are: "fullLocalRead" and "remoteRead". The first means that the input file is transferred on the node the SplitterPATask executes on before it begin to create input splits. The latter means that the input file is left out into the ProActive MapReduce workflow input space so that data used to create input splits are read directly from the ProActive MapReduce workflow input space. The default value is "remoteRead";
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.mapperPATask.readMode**: defines the read mode of the MapperPATask. There are three possible values:
  - **fullLocalRead**: the input file of the ProActive MapReduce workflow is transferred entirely on the node the MapperPATask executes on and then the MapperPATask read from it and elaborate only the data of its own input split;
  - **partialLocalRead**: only the data the MapperPATask must elaborate are copied from the input file and transferred on the node the MapperPATask executes on;
  - **remoteRead**: the data the MappperPATask must elaborate are read, remotely, from the input file that is left out into the ProActive MapReduce workflow input space.

  The default value is "remoteRead" but if the input file is not randomly accessible then "partialLocalRead" is used.
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.mapperPATask.writeMode**: defines the write mode of the MapperPATask. The two possible values are: "localWrite" and "remoteWrite". The former implies that the output data of the MapperPATask are first stored on the node it executes on and then the ProActive DataSpaces mechanism transfers them to the user defined output space. While the latter indicates that the output data are stored directly in the user defined output space. The default value is "localWrite";
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.reducerPATask.outputFileNamePrefix**: defines the prefix the ProActive MapReduce framework/API must use to build the name of the output file. The number of output files of the ProActive MapReduce job is equal to the number of executed ReducerPATask and each file has a name compliant to the following format: <outputFileNamePrefix>_<reducerPATaskId>;
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.reducerPATask.readMode**: defines the read mode of the ReducerPATask. The two possible value are: "fullLocalRead" and "remoteRead". The first means that the intermediate data (the MapperPATask output data) are transferred from the output space to the node the ReducerPATask will execute on while "remoteRead" means that the intermediate data are left in the output space so that they are read remotely by the ReducerPATask. The default value is "remoteRead";
- **org.ow2.proactive.scheduler.ext.mapreduce.workflow.reducerPATask.writeMode**: defines the write mode of the ReducerPATask. The two possible values are: "localWrite" and "remoteWrite". The former implies that the output data of the ReducerPATask are first stored on the node it executed on and then the ProActive DataSpaces mechanism transfers them to the user defined output space. While the latter indicates that the output data are stored directly in the user defined output space. The default value is "localWrite".

Lastly, the user has to notice that, in addition to specify the additional configuration parameters to build the ProActive MapReduce job using a property file, he can also specify them using the methods of the PAMapReduceJobConfiguration class.

10.3. ProActive MapReduce API restrictions

The most important restriction of the ProActive MapReduce framework/API is that it provides the support only for the **Hadoop MapReduce 0.20.2** release and only for the **Hadoop new MapReduce API** (this means that the Hadoop MapReduce job is built using the classes defined in the package **org.apache.hadoop.mapreduce**). Moreover, the ProActive MapReduce Hadoop-like API implementation does not support neither the execution of the Hadoop MapReduce **Combiner** class neither the data compression. This means if the user specify the class to execute as the combiner in the Hadoop MapReduce job, that class will be ignored and its methods

not executed by the ProActive MapReduce Hadoop-like API. The same will happen in the case of the compression/decompression mechanism the user could have choose to use for the execution of the Hadoop MapReduce job.

Part III. Appendice

Table of Contents

Chapter 11. XSD Job Schema

```
<?xml version="1.0" encoding="UTF-8"?><xs:schema xmlns:xs="http://www.w3.org/2001/
XMLSchema" xmlns:jd="urn:proactive:jobdescriptor:dev" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" elementFormDefault="qualified" targetNamespace="urn:proactive:jobdescriptor:dev">

 <xs:element name="job">
  <xs:annotation>
   <xs:documentation>Definition of a job for the scheduler </xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:sequence>
    <xs:element minOccurs="0" ref="jd:variables"/>
    <xs:group minOccurs="0" ref="jd:jobDescription"/>
    <xs:element minOccurs="0" ref="jd:jobClasspath"/>
    <xs:element minOccurs="0" ref="jd:genericInformation"/>
    <xs:element minOccurs="0" ref="jd:inputSpace"/>
    <xs:element minOccurs="0" ref="jd:outputSpace"/>
    <xs:element ref="jd:taskFlow"/>
   </xs:sequence>
   <xs:attributeGroup ref="jd:jobName"/>
   <xs:attribute name="priority">
    <xs:annotation>
     <xs:documentation>Priority of the job</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
     <xs:union memberTypes="jd:jobPriority jd:variableRefType"/>
    </xs:simpleType>
   </xs:attribute>
   <xs:attribute name="cancelJobOnError">
    <xs:annotation>
     <xs:documentation>For each task, does the job cancel right away if a single task had an error (by opposition to a
 network failure, memory error, etc...) (default=false)</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
     <xs:union memberTypes="xs:boolean jd:variableRefType"/>
    </xs:simpleType>
   </xs:attribute>
   <xs:attribute name="restartTaskOnError">
    <xs:annotation>
     <xs:documentation>For each task, where does the task restart if an error occurred ? (default=anywhere)</
xs:documentation>
    </xs:annotation>
    <xs:simpleType>
     <xs:union memberTypes="jd:restartTaskType jd:variableRefType"/>
    </xs:simpleType>
   </xs:attribute>
   <xs:attribute name="maxNumberOfExecution">
    <xs:annotation>
     <xs:documentation>Maximum number of execution for each task (default=1)</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
```

```
      <xs:union memberTypes="xs:nonNegativeInteger jd:variableRefType"/>
     </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="projectName" type="xs:string">
     <xs:annotation>
      <xs:documentation>Name of the project related to this job. It is also used in the policy to group some jobs
together</xs:documentation>
     </xs:annotation>
    </xs:attribute>
    <xs:attribute name="logFile">
     <xs:annotation>
      <xs:documentation>The path to a file where the logs of this job will be written </xs:documentation>
     </xs:annotation>
     <xs:simpleType>
      <xs:union memberTypes="xs:anyURI jd:variableRefType"/>
     </xs:simpleType>
    </xs:attribute>

  </xs:complexType>
 </xs:element>
 <xs:attributeGroup name="jobName">
  <xs:attribute name="name" use="required" type="xs:string">
   <xs:annotation>
    <xs:documentation>Identification of this job</xs:documentation>
   </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:element name="variables">
  <xs:annotation>
   <xs:documentation>Definition of variables which can be reused throughout this descriptor</xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:group maxOccurs="unbounded" ref="jd:variable"/>
  </xs:complexType>
 </xs:element>
 <xs:group name="variable">
  <xs:sequence>
   <xs:element name="variable">
    <xs:annotation>
     <xs:documentation>Definition of one variable, the variable can be reused (even in another following variable
definition) by using the syntax ${name_of_variable}</xs:documentation>
    </xs:annotation>
    <xs:complexType>
     <xs:attributeGroup ref="jd:variableName"/>
     <xs:attributeGroup ref="jd:variableValue"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:group>
 <xs:attributeGroup name="variableName">
  <xs:attribute name="name" use="required" type="xs:string">
   <xs:annotation>
    <xs:documentation>Name of a variable</xs:documentation>
   </xs:annotation>
```

```
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="variableValue">
  <xs:attribute name="value" use="required" type="xs:string">
   <xs:annotation>
    <xs:documentation>The patterns ${variable_name} will be replaced by this value</xs:documentation>
   </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="priority">
  <xs:attribute name="priority" use="required">
   <xs:annotation>
    <xs:documentation>Priority of the job</xs:documentation>
   </xs:annotation>
   <xs:simpleType>
    <xs:union memberTypes="jd:jobPriority jd:variableRefType"/>
   </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="cancelJobOnError_j">
  <xs:attribute name="cancelJobOnError" use="required">
   <xs:annotation>
    <xs:documentation>For each task, does the job cancel right away if a single task had an error (by opposition to a
 network failure, memory error, etc...) (default=false)</xs:documentation>
   </xs:annotation>
   <xs:simpleType>
    <xs:union memberTypes="xs:boolean jd:variableRefType"/>
   </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="restartTaskOnError_j">
  <xs:attribute name="restartTaskOnError" use="required">
   <xs:annotation>
    <xs:documentation>For each task, where does the task restart if an error occurred ? (default=anywhere)</
xs:documentation>
   </xs:annotation>
   <xs:simpleType>
    <xs:union memberTypes="jd:restartTaskType jd:variableRefType"/>
   </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="numberOfExecution_j">
  <xs:attribute name="maxNumberOfExecution" use="required">
   <xs:annotation>
    <xs:documentation>Maximum number of execution for each task (default=1)</xs:documentation>
   </xs:annotation>
   <xs:simpleType>
    <xs:union memberTypes="xs:nonNegativeInteger jd:variableRefType"/>
   </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="logFile">
  <xs:attribute name="logFile" use="required">
   <xs:annotation>
```

```
      <xs:documentation>The path to a file where the logs of this job will be written </xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:union memberTypes="xs:anyURI jd:variableRefType"/>
    </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:element name="jobClasspath">
  <xs:annotation>
    <xs:documentation>The classPath where to find the dependences of your job. It must contain path element</
xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="jd:pathElement"/>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="pathElement">
  <xs:annotation>
    <xs:documentation>Path element (one pathElement for each classpath entry)</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="path" use="required" type="xs:string"/>
  </xs:complexType>
 </xs:element>
 <xs:element name="genericInformation">
  <xs:annotation>
    <xs:documentation>Definition of any information you would like to get in the policy</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="jd:info"/>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="info">
  <xs:annotation>
    <xs:documentation>Information that you can get in the policy through the job content</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attributeGroup ref="jd:infoName"/>
    <xs:attributeGroup ref="jd:infoValue"/>
  </xs:complexType>
 </xs:element>
 <xs:attributeGroup name="infoName">
  <xs:attribute name="name" use="required" type="xs:string">
    <xs:annotation>
      <xs:documentation>Name of the information variable</xs:documentation>
    </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="infoValue">
  <xs:attribute name="value" use="required" type="xs:string">
```

```
    <xs:annotation>
      <xs:documentation>Value of the information variable</xs:documentation>
    </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="projectName">
  <xs:attribute name="projectName" use="required" type="xs:string">
    <xs:annotation>
      <xs:documentation>Name of the project related to this job. It is also used in the policy to group some jobs
together</xs:documentation>
    </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:group name="jobDescription">
  <xs:sequence>
    <xs:element name="description" type="xs:string">
      <xs:annotation>
        <xs:documentation>Textual description of this job </xs:documentation>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
 </xs:group>
 <xs:element name="taskFlow">
  <xs:annotation>
    <xs:documentation>A job composed of a flow of tasks with or without dependencies </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:group maxOccurs="unbounded" ref="jd:task"/>
  </xs:complexType>
 </xs:element>
 <!-- ++++++++++++ task -->
 <xs:group name="task">
  <xs:sequence>
    <xs:element name="task">
      <xs:annotation>
        <xs:documentation>A task is the smallest computation unit for the scheduler </xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:sequence>
          <xs:group minOccurs="0" ref="jd:taskDescription"/>
          <xs:element minOccurs="0" ref="jd:genericInformation"/>
          <xs:element minOccurs="0" ref="jd:depends"/>
          <xs:element minOccurs="0" ref="jd:inputFiles"/>
          <xs:element minOccurs="0" ref="jd:parallel"/>
          <xs:element minOccurs="0" ref="jd:selection"/>
          <xs:element minOccurs="0" ref="jd:pre"/>
          <xs:element ref="jd:executable"/>
          <xs:element minOccurs="0" ref="jd:controlFlow"/>
          <xs:element minOccurs="0" ref="jd:post"/>
          <xs:element minOccurs="0" ref="jd:cleaning"/>
          <xs:element minOccurs="0" ref="jd:outputFiles"/>
        </xs:sequence>
        <xs:attributeGroup ref="jd:taskName"/>
        <xs:attribute name="numberOfNodes">
```

```
        <xs:annotation>
          <xs:documentation>number of cores needed for the task (identifier)</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:union memberTypes="xs:positiveInteger jd:variableRefType"/>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="cancelJobOnError">
        <xs:annotation>
          <xs:documentation>Does the job cancel right away if a single task had an error (by opposition to a network
failure, memory error, etc...)</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:union memberTypes="xs:boolean jd:variableRefType"/>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="restartTaskOnError" type="jd:restartTaskType">
        <xs:annotation>
          <xs:documentation>Where does the task restart if an error occurred ?</xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="maxNumberOfExecution">
        <xs:annotation>
          <xs:documentation>Maximum number of execution for this task</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:union memberTypes="xs:nonNegativeInteger jd:variableRefType"/>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="runAsMe">
        <xs:annotation>
          <xs:documentation>Do we run this task under the job owner user identity</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:union memberTypes="xs:boolean jd:variableRefType"/>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="walltime">
        <xs:annotation>
          <xs:documentation>Defines walltime - maximum execution time of the task. (patterns are 'ss' OR 'mm:ss' OR
'hh:mm:ss')</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:union memberTypes="jd:walltimePattern jd:variableRefType"/>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="preciousResult">
        <xs:annotation>
          <xs:documentation>Do we keep the result among the job results  </xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:union memberTypes="xs:boolean jd:variableRefType"/>
        </xs:simpleType>
      </xs:attribute>
```

```
      <xs:attribute name="resultPreviewClass">
        <xs:annotation>
          <xs:documentation>A class implementing the ResultPreview interface which can be used to display the result
of this task</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:union memberTypes="jd:classPattern jd:variableRefType"/>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:group>
<xs:attributeGroup name="taskName">
 <xs:attribute name="name" use="required" type="xs:ID">
  <xs:annotation>
   <xs:documentation>Identification of this task (identifier) </xs:documentation>
  </xs:annotation>
 </xs:attribute>
</xs:attributeGroup>
<xs:attributeGroup name="nodesNumber_t">
 <xs:attribute name="numberOfNodes" use="required">
  <xs:annotation>
   <xs:documentation>number of cores needed for the task (identifier)</xs:documentation>
  </xs:annotation>
  <xs:simpleType>
   <xs:union memberTypes="xs:positiveInteger jd:variableRefType"/>
  </xs:simpleType>
 </xs:attribute>
</xs:attributeGroup>
<xs:group name="taskDescription">
 <xs:sequence>
  <xs:element name="description" type="xs:string">
   <xs:annotation>
    <xs:documentation>Textual description of this task</xs:documentation>
   </xs:annotation>
  </xs:element>
 </xs:sequence>
</xs:group>
<xs:attributeGroup name="walltime">
 <xs:attribute name="walltime" use="required">
  <xs:annotation>
   <xs:documentation>Defines walltime - maximum execution time of the task. (patterns are 'ss' OR 'mm:ss' OR
'hh:mm:ss')</xs:documentation>
  </xs:annotation>
  <xs:simpleType>
   <xs:union memberTypes="jd:walltimePattern jd:variableRefType"/>
  </xs:simpleType>
 </xs:attribute>
</xs:attributeGroup>
<xs:attributeGroup name="cancelJobOnError_t">
 <xs:attribute name="cancelJobOnError" use="required">
  <xs:annotation>
```

```
      <xs:documentation>Does the job cancel right away if a single task had an error (by opposition to a network failure, memory error, etc...)</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:union memberTypes="xs:boolean jd:variableRefType"/>
    </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="restartTaskOnError_t">
  <xs:attribute name="restartTaskOnError" use="required" type="jd:restartTaskType">
    <xs:annotation>
      <xs:documentation>Where does the task restart if an error occurred ?</xs:documentation>
    </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="numberOfExecution_t">
  <xs:attribute name="maxNumberOfExecution" use="required">
    <xs:annotation>
      <xs:documentation>Maximum number of execution for this task</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:union memberTypes="xs:nonNegativeInteger jd:variableRefType"/>
    </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="preciousResult">
  <xs:attribute name="preciousResult" use="required">
    <xs:annotation>
      <xs:documentation>Do we keep the result among the job results  </xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:union memberTypes="xs:boolean jd:variableRefType"/>
    </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="resultPreviewClass">
  <xs:attribute name="resultPreviewClass" use="required">
    <xs:annotation>
      <xs:documentation>A class implementing the ResultPreview interface which can be used to display the result of this task</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:union memberTypes="jd:classPattern jd:variableRefType"/>
    </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:element name="depends">
  <xs:annotation>
    <xs:documentation>A list of dependencies for this task </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:group maxOccurs="unbounded" ref="jd:dependsTask"/>
  </xs:complexType>
 </xs:element>
```

```
<xs:group name="dependsTask">
 <xs:sequence>
  <xs:element name="task">
   <xs:annotation>
    <xs:documentation>A task from which this task depends </xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:attribute name="ref" use="required" type="xs:IDREF"/>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:group>
<xs:element name="parallel">
 <xs:annotation>
  <xs:documentation>An information related parallel task including nodes number and topology</xs:documentation>
 </xs:annotation>
 <xs:complexType>
  <xs:sequence>
   <xs:element minOccurs="0" ref="jd:topology"/>
  </xs:sequence>
  <xs:attributeGroup ref="jd:nodesNumber_t"/>
 </xs:complexType>
</xs:element>
<xs:element name="topology">
 <xs:annotation>
  <xs:documentation>A topology descriptor of the parallel task</xs:documentation>
 </xs:annotation>
 <xs:complexType>
  <xs:choice>
   <xs:element ref="jd:arbitrary"/>
   <xs:element ref="jd:bestProximity"/>
   <xs:element ref="jd:thresholdProximity"/>
   <xs:element ref="jd:singleHost"/>
   <xs:element ref="jd:singleHostExclusive"/>
   <xs:element ref="jd:multipleHostsExclusive"/>
   <xs:element ref="jd:differentHostsExclusive"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
<xs:element name="selection">
 <xs:annotation>
  <xs:documentation>A script used to select resources that can handle the task </xs:documentation>
 </xs:annotation>
 <xs:complexType>
  <xs:group maxOccurs="unbounded" ref="jd:selectionScript"/>
 </xs:complexType>
</xs:element>
<xs:element name="controlFlow">
 <xs:annotation>
  <xs:documentation>Flow control : block declaration, flow action</xs:documentation>
 </xs:annotation>
 <xs:complexType>
  <xs:choice minOccurs="0">
   <xs:element ref="jd:if"/>
```

```
      <xs:element ref="jd:replicate"/>
      <xs:element ref="jd:loop"/>
    </xs:choice>
    <xs:attribute name="block" type="jd:blockAttr">
      <xs:annotation>
        <xs:documentation>block declaration</xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:attributeGroup name="block">
  <xs:attribute name="block" use="required" type="jd:blockAttr">
    <xs:annotation>
      <xs:documentation>block declaration</xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:attributeGroup>
<xs:element name="pre" type="jd:script">
  <xs:annotation>
    <xs:documentation>A script launched before the task execution in the task node</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="post" type="jd:script">
  <xs:annotation>
    <xs:documentation>A script launched after the task execution in the task node</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="cleaning" type="jd:script">
  <xs:annotation>
    <xs:documentation>A script launched by the Resource Manager after the task or post script</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:attributeGroup name="runAsMe">
  <xs:attribute name="runAsMe" use="required">
    <xs:annotation>
      <xs:documentation>Do we run this task under the job owner user identity</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:union memberTypes="xs:boolean jd:variableRefType"/>
    </xs:simpleType>
  </xs:attribute>
</xs:attributeGroup>
<!-- ++++++++++++ scripts -->
<xs:complexType name="script">
  <xs:sequence>
    <xs:element name="script">
      <xs:annotation>
        <xs:documentation>Definition of a standard script </xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:choice>
          <xs:element ref="jd:code"/>
          <xs:element ref="jd:file"/>
        </xs:choice>
```

```
    </xs:complexType>
   </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:group name="selectionScript">
  <xs:sequence>
   <xs:element name="script">
    <xs:annotation>
     <xs:documentation>Definition of a specific script which is used for resource selection </xs:documentation>
    </xs:annotation>
    <xs:complexType>
     <xs:choice>
      <xs:element ref="jd:code"/>
      <xs:element ref="jd:file"/>
     </xs:choice>
     <xs:attribute name="type">
      <xs:annotation>
       <xs:documentation>Type of script for the infrastructure manager (default to dynamic)  </xs:documentation>
      </xs:annotation>
      <xs:simpleType>
       <xs:restriction base="xs:token">
        <xs:enumeration value="dynamic"/>
        <xs:enumeration value="static"/>
       </xs:restriction>
      </xs:simpleType>
     </xs:attribute>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
</xs:group>
<xs:attributeGroup name="scriptType">
  <xs:attribute name="type" use="required">
   <xs:annotation>
    <xs:documentation>Type of script for the infrastructure manager (default to dynamic)  </xs:documentation>
   </xs:annotation>
   <xs:simpleType>
    <xs:restriction base="xs:token">
     <xs:enumeration value="dynamic"/>
     <xs:enumeration value="static"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
</xs:attributeGroup>
<xs:element name="code">
  <xs:annotation>
   <xs:documentation>Definition of a script by writing directly the code inside the descriptor </xs:documentation>
  </xs:annotation>
  <xs:complexType mixed="true">
   <xs:attribute name="language" use="required" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:element name="file">
  <xs:annotation>
   <xs:documentation>Definition of a script by loading a file </xs:documentation>
```

```
  </xs:annotation>
  <xs:complexType>
    <xs:group minOccurs="0" ref="jd:fileScriptArguments"/>
    <xs:attribute name="path">
      <xs:annotation>
        <xs:documentation>File path to script definition  </xs:documentation>
      </xs:annotation>
      <xs:simpleType>
        <xs:union memberTypes="xs:anyURI jd:variableRefType"/>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="url">
      <xs:annotation>
        <xs:documentation>Remote script definition, reachable at the given url </xs:documentation>
      </xs:annotation>
      <xs:simpleType>
        <xs:union memberTypes="xs:anyURI jd:variableRefType"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:attributeGroup name="path">
  <xs:attribute name="path" use="required">
    <xs:annotation>
      <xs:documentation>File path to script definition  </xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:union memberTypes="xs:anyURI jd:variableRefType"/>
    </xs:simpleType>
  </xs:attribute>
</xs:attributeGroup>
<xs:attributeGroup name="url">
  <xs:attribute name="url" use="required">
    <xs:annotation>
      <xs:documentation>Remote script definition, reachable at the given url </xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:union memberTypes="xs:anyURI jd:variableRefType"/>
    </xs:simpleType>
  </xs:attribute>
</xs:attributeGroup>
<xs:group name="fileScriptArguments">
  <xs:sequence>
    <xs:element name="arguments">
      <xs:annotation>
        <xs:documentation>A list of arguments of this script </xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:group maxOccurs="unbounded" ref="jd:fileScriptArgument"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:group>
<xs:group name="fileScriptArgument">
```

```
  <xs:sequence>
    <xs:element name="argument">
      <xs:annotation>
        <xs:documentation>An argument of this script  </xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:attribute name="value" use="required" type="xs:string"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:group>
<!-- +++++++++++++ Topology -->
<xs:element name="arbitrary">
  <xs:annotation>
    <xs:documentation>arbitrary topology</xs:documentation>
  </xs:annotation>
  <xs:complexType/>
</xs:element>
<xs:element name="bestProximity">
  <xs:annotation>
    <xs:documentation>best nodes proximity</xs:documentation>
  </xs:annotation>
  <xs:complexType/>
</xs:element>
<xs:element name="thresholdProximity">
  <xs:annotation>
    <xs:documentation>threshold nodes proximity</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attributeGroup ref="jd:threshold"/>
  </xs:complexType>
</xs:element>
<xs:element name="singleHost">
  <xs:annotation>
    <xs:documentation>nodes on single host</xs:documentation>
  </xs:annotation>
  <xs:complexType/>
</xs:element>
<xs:element name="singleHostExclusive">
  <xs:annotation>
    <xs:documentation>nodes on single host exclusively</xs:documentation>
  </xs:annotation>
  <xs:complexType/>
</xs:element>
<xs:element name="multipleHostsExclusive">
  <xs:annotation>
    <xs:documentation>nodes on multiple hosts exclusively</xs:documentation>
  </xs:annotation>
  <xs:complexType/>
</xs:element>
<xs:element name="differentHostsExclusive">
  <xs:annotation>
    <xs:documentation>nodes on single host exclusively</xs:documentation>
  </xs:annotation>
```

```
  <xs:complexType/>
 </xs:element>
 <xs:attributeGroup name="threshold">
  <xs:attribute name="threshold" use="required" type="xs:nonNegativeInteger">
   <xs:annotation>
    <xs:documentation>threshold value</xs:documentation>
   </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <!-- ++++++++++++ executables -->
 <xs:element name="executable" abstract="true"/>
 <xs:element name="nativeExecutable" substitutionGroup="jd:executable">
  <xs:annotation>
   <xs:documentation>A native command call, it can be statically described or generated by a script </
xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:choice>
    <xs:element ref="jd:staticCommand"/>
    <xs:element ref="jd:dynamicCommand"/>
   </xs:choice>
  </xs:complexType>
 </xs:element>
 <xs:element name="staticCommand">
  <xs:annotation>
   <xs:documentation>A native command statically defined in the descriptor </xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:group minOccurs="0" ref="jd:commandArguments"/>
   <xs:attribute name="value" use="required" type="xs:string"/>
   <xs:attribute name="workingDir" type="xs:string">
    <xs:annotation>
     <xs:documentation>working dir for native command to execute (ie launching dir, pwd...)</xs:documentation>
    </xs:annotation>
   </xs:attribute>
  </xs:complexType>
 </xs:element>
 <xs:attributeGroup name="workingDir_t">
  <xs:attribute name="workingDir" use="required" type="xs:string">
   <xs:annotation>
    <xs:documentation>working dir for native command to execute (ie launching dir, pwd...)</xs:documentation>
   </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:group name="commandArguments">
  <xs:sequence>
   <xs:element name="arguments">
    <xs:annotation>
     <xs:documentation>List of arguments to the native command (they will be appended at the end of the command)
</xs:documentation>
    </xs:annotation>
    <xs:complexType>
     <xs:group maxOccurs="unbounded" ref="jd:commandArgument"/>
    </xs:complexType>
```

```
    </xs:element>
  </xs:sequence>
</xs:group>
<xs:group name="commandArgument">
  <xs:sequence>
    <xs:element name="argument">
      <xs:complexType>
        <xs:attribute name="value" use="required" type="xs:string"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:group>
<xs:element name="dynamicCommand">
  <xs:annotation>
    <xs:documentation>A command generated dynamically </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="jd:generation">
        <xs:attribute name="workingDir" type="xs:string">
          <xs:annotation>
            <xs:documentation>working dir for native command to execute (ie launching dir, pwd...)</xs:documentation>
          </xs:annotation>
        </xs:attribute>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:complexType name="generation">
  <xs:sequence>
    <xs:element ref="jd:generation"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="generation" type="jd:script">
  <xs:annotation>
    <xs:documentation>A command generated by a script </xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="javaExecutable" substitutionGroup="jd:executable">
  <xs:annotation>
    <xs:documentation>A Java class implementing the Executable interface </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" ref="jd:forkEnvironment"/>
      <xs:element minOccurs="0" ref="jd:parameters"/>
    </xs:sequence>
    <xs:attributeGroup ref="jd:class"/>
  </xs:complexType>
</xs:element>
<xs:attributeGroup name="class">
  <xs:attribute name="class" use="required">
    <xs:annotation>
      <xs:documentation>The fully qualified class name </xs:documentation>
```

```
    </xs:annotation>
    <xs:simpleType>
      <xs:union memberTypes="jd:classPattern jd:variableRefType"/>
    </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:element name="forkEnvironment">
  <xs:annotation>
    <xs:documentation>Fork environment if needed</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" ref="jd:SystemEnvironment"/>
      <xs:element minOccurs="0" ref="jd:jvmArgs"/>
      <xs:element minOccurs="0" ref="jd:additionalClasspath"/>
      <xs:element minOccurs="0" ref="jd:envScript"/>
    </xs:sequence>
    <xs:attribute name="workingDir" type="xs:string">
      <xs:annotation>
        <xs:documentation>working dir for native command to execute (ie launching dir, pwd...)</xs:documentation>
      </xs:annotation>
    </xs:attribute>
    <xs:attribute name="javaHome">
      <xs:annotation>
        <xs:documentation>Path to installed java directory, to this path '/bin/java' will be added, if attribute does not exist
only 'java' command will be called</xs:documentation>
      </xs:annotation>
      <xs:simpleType>
        <xs:union memberTypes="xs:anyURI jd:variableRefType"/>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
 </xs:element>
 <xs:attributeGroup name="javaHome">
  <xs:attribute name="javaHome" use="required">
    <xs:annotation>
      <xs:documentation>Path to installed java directory, to this path '/bin/java' will be added, if attribute does not exist
only 'java' command will be called</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:union memberTypes="xs:anyURI jd:variableRefType"/>
    </xs:simpleType>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:element name="SystemEnvironment">
  <xs:annotation>
    <xs:documentation>a list of sysProp</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:group maxOccurs="unbounded" ref="jd:sysProp"/>
  </xs:complexType>
 </xs:element>
 <xs:group name="sysProp">
  <xs:sequence>
```

```
    <xs:element name="variable">
     <xs:annotation>
      <xs:documentation>A parameter in the form of key/value pair</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:attribute name="name" use="required" type="xs:string"/>
      <xs:attribute name="value" use="required" type="xs:string"/>
      <xs:attribute name="append">
       <xs:annotation>
        <xs:documentation>append or not this property to a previous declared or to the system one</
xs:documentation>
       </xs:annotation>
       <xs:simpleType>
        <xs:union memberTypes="xs:boolean jd:variableRefType"/>
       </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="appendChar" type="jd:propertyAppendChar">
       <xs:annotation>
        <xs:documentation>append character, if set, append mode is ON</xs:documentation>
       </xs:annotation>
      </xs:attribute>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:group>
  <xs:attributeGroup name="sysPropAppend">
   <xs:attribute name="append" use="required">
    <xs:annotation>
     <xs:documentation>append or not this property to a previous declared or to the system one</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
     <xs:union memberTypes="xs:boolean jd:variableRefType"/>
    </xs:simpleType>
   </xs:attribute>
  </xs:attributeGroup>
  <xs:attributeGroup name="sysPropAppendChar">
   <xs:attribute name="appendChar" use="required" type="jd:propertyAppendChar">
    <xs:annotation>
     <xs:documentation>append character, if set, append mode is ON</xs:documentation>
    </xs:annotation>
   </xs:attribute>
  </xs:attributeGroup>
  <xs:element name="jvmArgs">
   <xs:annotation>
    <xs:documentation>A list of java properties or options</xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:sequence>
     <xs:element maxOccurs="unbounded" ref="jd:jvmArg"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="jvmArg">
   <xs:annotation>
```

```
    <xs:documentation>A list of java properties or options</xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:attribute name="value" use="required" type="xs:string"/>
   </xs:complexType>
  </xs:element>
  <xs:element name="additionalClasspath">
   <xs:annotation>
    <xs:documentation>classpath entries to add to the new java process</xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:sequence>
     <xs:element maxOccurs="unbounded" ref="jd:pathElement"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="envScript" type="jd:script">
   <xs:annotation>
    <xs:documentation>environment script to execute for setting forked process environment</xs:documentation>
   </xs:annotation>
  </xs:element>
  <xs:element name="parameters">
   <xs:annotation>
    <xs:documentation>A list of parameters that will be given to the Java task through the init method </
xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:sequence>
     <xs:element maxOccurs="unbounded" ref="jd:parameter"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="parameter">
   <xs:annotation>
    <xs:documentation>A parameter in the form of key/value pair </xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:attribute name="name" use="required" type="xs:string"/>
    <xs:attribute name="value" use="required" type="xs:string"/>
   </xs:complexType>
  </xs:element>
  <!-- ++++++++++++ DataSpaces -->
  <xs:element name="inputSpace">
   <xs:annotation>
    <xs:documentation>Input space for the job, URL that define root directory containing job needed files</
xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:attributeGroup ref="jd:url"/>
   </xs:complexType>
  </xs:element>
  <xs:element name="outputSpace">
   <xs:annotation>
```

```
    <xs:documentation>Output space for the job, URL that define root directory that will contain job generated output
 files</xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:attributeGroup ref="jd:url"/>
   </xs:complexType>
  </xs:element>
  <xs:element name="inputFiles">
   <xs:annotation>
    <xs:documentation>selection of input files that will be accessible for the application and copied from INPUT/
OUTPUT to local system</xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:group maxOccurs="unbounded" ref="jd:infiles_"/>
   </xs:complexType>
  </xs:element>
  <xs:group name="infiles_">
   <xs:sequence>
    <xs:element name="files">
     <xs:annotation>
      <xs:documentation>description of input files with include, exclude and an access mode</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:attributeGroup ref="jd:includes_"/>
      <xs:attribute name="excludes" type="jd:inexcludePattern">
       <xs:annotation>
        <xs:documentation>Pattern of the files to exclude among the included one, relative to INPUT or OUTPUT
 spaces</xs:documentation>
       </xs:annotation>
      </xs:attribute>
      <xs:attributeGroup ref="jd:inaccessMode_"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:group>
  <xs:attributeGroup name="inaccessMode_">
   <xs:attribute name="accessMode" use="required" type="jd:inaccessModeType">
    <xs:annotation>
     <xs:documentation>type of access on the selected files</xs:documentation>
    </xs:annotation>
   </xs:attribute>
  </xs:attributeGroup>
  <xs:element name="outputFiles">
   <xs:annotation>
    <xs:documentation>selection of output files that will be copied from LOCALSPACE to OUTPUT</
xs:documentation>
   </xs:annotation>
   <xs:complexType>
    <xs:group maxOccurs="unbounded" ref="jd:outfiles_"/>
   </xs:complexType>
  </xs:element>
  <xs:group name="outfiles_">
   <xs:sequence>
    <xs:element name="files">
```

```
      <xs:annotation>
        <xs:documentation>description of output files with include, exclude and an access mode</xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:attributeGroup ref="jd:includes_"/>
        <xs:attribute name="excludes" type="jd:inexcludePattern">
          <xs:annotation>
            <xs:documentation>Pattern of the files to exclude among the included one, relative to INPUT or OUTPUT
spaces</xs:documentation>
          </xs:annotation>
        </xs:attribute>
        <xs:attributeGroup ref="jd:outaccessMode_"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
 </xs:group>
 <xs:attributeGroup name="outaccessMode_">
  <xs:attribute name="accessMode" use="required" type="jd:outaccessModeType">
    <xs:annotation>
      <xs:documentation>type of access on the selected files</xs:documentation>
    </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="includes_">
  <xs:attribute name="includes" use="required" type="jd:inexcludePattern">
    <xs:annotation>
      <xs:documentation>Pattern of the files to include, relative to INPUT or OUTPUT spaces</xs:documentation>
    </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <xs:attributeGroup name="excludes_">
  <xs:attribute name="excludes" use="required" type="jd:inexcludePattern">
    <xs:annotation>
      <xs:documentation>Pattern of the files to exclude among the included one, relative to INPUT or OUTPUT
spaces</xs:documentation>
    </xs:annotation>
  </xs:attribute>
 </xs:attributeGroup>
 <!-- ++++++++++++ Control flow -->
 <xs:element name="if">
  <xs:annotation>
    <xs:documentation>branching Control flow action</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="jd:script">
        <xs:attributeGroup ref="jd:target"/>
        <xs:attributeGroup ref="jd:targetElse"/>
        <xs:attribute name="continuation" type="xs:string">
          <xs:annotation>
            <xs:documentation>branching else target</xs:documentation>
          </xs:annotation>
        </xs:attribute>
      </xs:extension>
```

```
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:attributeGroup name="target">
  <xs:attribute name="target" use="required" type="xs:string">
    <xs:annotation>
      <xs:documentation>branching if target</xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:attributeGroup>
<xs:attributeGroup name="targetElse">
  <xs:attribute name="else" use="required" type="xs:string">
    <xs:annotation>
      <xs:documentation>branching else target</xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:attributeGroup>
<xs:attributeGroup name="targetContinuation">
  <xs:attribute name="continuation" use="required" type="xs:string">
    <xs:annotation>
      <xs:documentation>branching else target</xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:attributeGroup>
<xs:element name="loop">
  <xs:annotation>
    <xs:documentation>looping control flow action</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="jd:script">
        <xs:attributeGroup ref="jd:target"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name="replicate" type="jd:script">
  <xs:annotation>
    <xs:documentation>replicate control flow action</xs:documentation>
  </xs:annotation>
</xs:element>
<!-- ++++++++++++ User define types -->
<xs:simpleType name="jobPriority">
  <xs:union memberTypes="jd:variableRefType">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="highest"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="high"/>
      </xs:restriction>
    </xs:simpleType>
```

```
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="normal"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="low"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="lowest"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
<xs:simpleType name="restartTaskType">
  <xs:union memberTypes="jd:variableRefType">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="anywhere"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="elsewhere"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
<xs:simpleType name="inaccessModeType">
  <xs:union memberTypes="jd:variableRefType">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="transferFromInputSpace"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="transferFromOutputSpace"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="transferFromGlobalSpace"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="none"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
```

```
</xs:simpleType>
<xs:simpleType name="outaccessModeType">
 <xs:union memberTypes="jd:variableRefType">
  <xs:simpleType>
   <xs:restriction base="xs:token">
    <xs:enumeration value="transferToOutputSpace"/>
   </xs:restriction>
  </xs:simpleType>
  <xs:simpleType>
   <xs:restriction base="xs:token">
    <xs:enumeration value="transferToGlobalSpace"/>
   </xs:restriction>
  </xs:simpleType>
  <xs:simpleType>
   <xs:restriction base="xs:token">
    <xs:enumeration value="none"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:union>
</xs:simpleType>
<xs:simpleType name="classPattern">
 <xs:restriction base="xs:string">
  <xs:pattern value="([A-Za-z_$][A-Za-z_0-9$]*\.)*[A-Za-z_$][A-Za-z_0-9$]*"/>
 </xs:restriction>
</xs:simpleType>
<xs:simpleType name="walltimePattern">
 <xs:restriction base="xs:string">
  <xs:pattern value="[0-9]*[1-9][0-9]*(:[0-5][0-9]){0,2}"/>
 </xs:restriction>
</xs:simpleType>
<xs:simpleType name="variableRefType">
 <xs:restriction base="xs:string">
  <xs:pattern value="$\{[A-Za-z0-9._]+\}"/>
 </xs:restriction>
</xs:simpleType>
<xs:simpleType name="inexcludePattern">
 <xs:restriction base="xs:string">
  <xs:pattern value=".+(,.+)*"/>
 </xs:restriction>
</xs:simpleType>
<xs:simpleType name="controlFlowAction">
 <xs:restriction base="xs:token">
  <xs:enumeration value="goto"/>
  <xs:enumeration value="replicate"/>
  <xs:enumeration value="continue"/>
 </xs:restriction>
</xs:simpleType>
<xs:simpleType name="blockAttr">
 <xs:restriction base="xs:token">
  <xs:enumeration value="start"/>
  <xs:enumeration value="end"/>
  <xs:enumeration value="none"/>
 </xs:restriction>
</xs:simpleType>
```

```
 <xs:simpleType name="propertyAppendChar">
  <xs:restriction base="xs:string">
   <xs:pattern value="."/>
  </xs:restriction>
 </xs:simpleType>
</xs:schema>
```