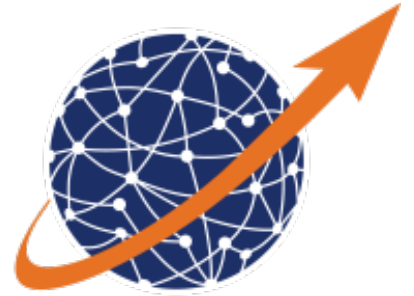


# ProActive Parallel Suite

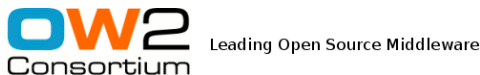


An Open Source Middleware For Parallel, Distributed, Multicore Computing

## ProActive Parallel Services

Version 1.0.0

The OASIS Research Team and ActiveEon Company





# ProActive Parallel Services v1.0.0 Documentation

## Legal Notice

This library is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; version 3 of the License.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

If needed, contact us to obtain a release under GPL Version 2, or a different license than the GPL.

Contact: [proactive@ow2.org](mailto:proactive@ow2.org) or [contact@activeeon.com](mailto:contact@activeeon.com)

Copyright 1997-2010 INRIA/University of Nice-Sophia Antipolis/ActiveEon.

## Mailing List

[proactive@ow2.org](mailto: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](mailto:Denis.Caromel@inria.fr)

---

### Contributors from OASIS Team

Brian Amedro  
Francoise Baude  
Francesco Bongiovanni  
Florin-Alexandru Bratu  
Viet Dung Doan  
Yu Feng  
Imen Filali  
Fabrice Fontenoy  
Ludovic Henrio  
Fabrice Huet  
Elaine Isnard  
Vasile Jureschi  
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

---

### Contributors from ActiveEon Company

Vladimir Bodnartchouk  
Arnaud Contes  
Cédric Dalmasso  
Christian Delbé  
Arnaud Gastinel  
Jean-Michel Guillaume  
Olivier Helin  
Clément Mathieu  
Maxime Menant  
Emil Salageanu  
Jean-Luc Scheefer  
Mathieu Schnoor

---

### 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 ..... iii

## Part I. ProActive Parallel Services

**Chapter 1. Overview** ..... 2

- 1.1. Introduction ..... 2
- 1.2. A Service Oriented / Grid Infrastructure ..... 2
- 1.3. The Parallel Services set ..... 4

**Chapter 2. Functional Description of Parallel Services** ..... 5

- 2.1. PSRepository: a repository for Parallel Services user resources ..... 5
  - 2.1.1. Overview ..... 5
  - 2.1.2. The PSRepository functionalities ..... 5
- 2.2. SchedulerInterface: a Web Service interface for ProActive Scheduler ..... 6
  - 2.2.1. Overview ..... 6
  - 2.2.2. The SchedulerInterface functionalities ..... 6
- 2.3. ParameterSweeping: a parallel computation pattern running jobs on the ProActive Scheduler. .... 14
  - 2.3.1. Parallel Pattern Services ..... 14
  - 2.3.2. Parameter Sweeping Overview ..... 15
  - 2.3.3. Parameter Sweeping HotSpots - User Specific Implementation ..... 15
  - 2.3.4. The ParameterSweeping Functionalities ..... 18
- 2.4. Security requirements ..... 20

**Chapter 3. Implementation details** ..... 21

- 3.1. Overview ..... 21
- 3.2. PSRepository implementation details ..... 21
- 3.3. SchedulerInterface implementation details ..... 23
- 3.4. Parameter Sweeping implementation details ..... 24

**Chapter 4. Installation** ..... 28

- 4.1. Introduction ..... 28
- 4.2. Installation of the infrastructure ..... 28
- 4.3. Web Services Configuration and Deployment ..... 28
- 4.4. Run the Parallel Services ..... 31
- 4.5. Web Service building from sources ..... 31

# List of Figures

1.1. An high level view of a SOA / GRID infrastructure .....	3
1.2. Parallel Services overview .....	4
2.1. PSRepository Web Service interface .....	6
2.2. Job archive structure .....	8
2.3. SchedulerInterface interface (partial view) .....	14
2.4. ParameterSweeping Hot Spots .....	16
2.5. ParameterSweeping - passing arguments between different entities .....	18
2.6. ParameterSweeping interface .....	20
3.1. Parallel Services subsystem .....	21
3.2. PSRepository main packages .....	22
3.3. Repository Interface API .....	22
3.4. Repository Interface data base implementation .....	23
3.5. SchedulerInterface main packages .....	23
3.6. Job Submission via archive .....	24
3.7. Parameter Sweeping main packages .....	25
3.8. Parameter Sweeping invocation scenario - asynchronous .....	26

# Part I. ProActive Parallel Services

## Table of Contents

<b>Chapter 1. Overview</b> .....	<b>2</b>
1.1. Introduction .....	2
1.2. A Service Oriented / Grid Infrastructure .....	2
1.3. The Parallel Services set .....	4
<b>Chapter 2. Functional Description of Parallel Services</b> .....	<b>5</b>
2.1. PSRepository: a repository for Parallel Services user resources .....	5
2.1.1. Overview .....	5
2.1.2. The PSRepository functionalities .....	5
2.2. SchedulerInterface: a Web Service interface for ProActive Scheduler .....	6
2.2.1. Overview .....	6
2.2.2. The SchedulerInterface functionalities .....	6
2.3. ParameterSweeping: a parallel computation pattern running jobs on the ProActive Scheduler. ....	14
2.3.1. Parallel Pattern Services .....	14
2.3.2. Parameter Sweeping Overview .....	15
2.3.3. Parameter Sweeping HotSpots - User Specific Implementation .....	15
2.3.4. The ParameterSweeping Functionalities .....	18
2.4. Security requirements .....	20
<b>Chapter 3. Implementation details</b> .....	<b>21</b>
3.1. Overview .....	21
3.2. PSRepository implementation details .....	21
3.3. SchedulerInterface implementation details .....	23
3.4. Parameter Sweeping implementation details .....	24
<b>Chapter 4. Installation</b> .....	<b>28</b>
4.1. Introduction .....	28
4.2. Installation of the infrastructure .....	28
4.3. Web Services Configuration and Deployment .....	28
4.4. Run the Parallel Services .....	31
4.5. Web Service building from sources .....	31

# Chapter 1. Overview

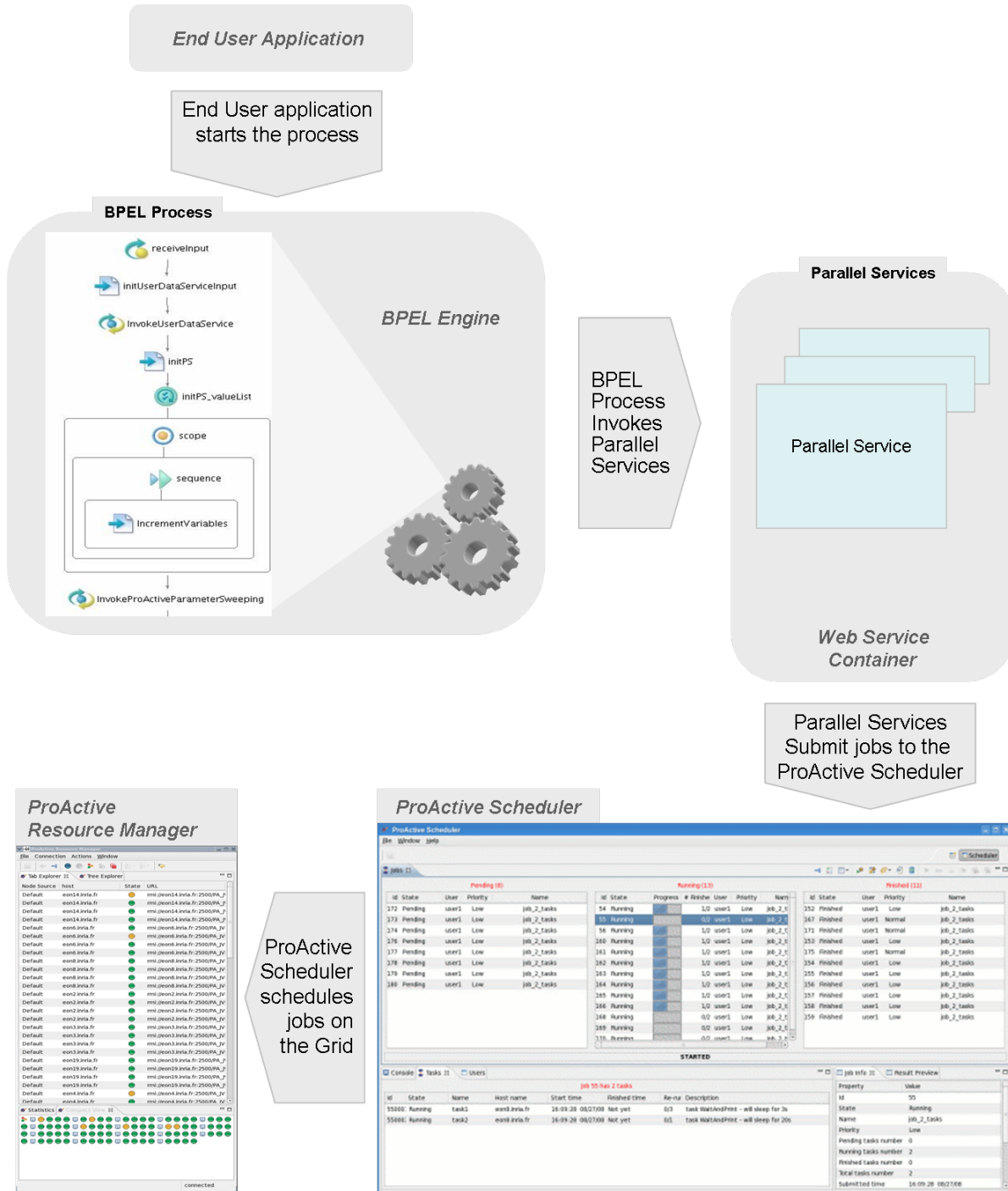
## 1.1. Introduction

Parallel Services are a set of Web Services that aims to represent an access point to the ProActive Scheduler from outside of the Grid infrastructure domain where the Scheduler is installed in. There are essentially two main needs that motivates the introduction of these services: from the interoperability point of view a Web Service standard access to the ProActive Scheduler enables or simplifies users running in a specific administrative domain to take advantage of a Grid infrastructure deployed and managed in a separate administrative domain; from the system integration and software reusing point of view, exposing parallel mechanisms as services is a way to be able to reuse and compose them in a standard workflow language. The last aspect is quite important in a Service Oriented approach, where the main goal is to build a collection of services that hides the complexity of implementation details and that may be reused to easily build new services increasing organization agility.

Basically a Parallel Service is a service potentially executed in parallel on several machines, since its logic is dynamically mapped onto hosts by the ProActive Scheduler.

## 1.2. A Service Oriented / Grid Infrastructure

In the context of a Service Oriented Architecture (SOA), the Grid infrastructure exposes Parallel Services to external users, that are part of the SOA domain, in order to let them reuse Grid functionalities in an easy way. Functionalities exposed by the Grid infrastructure are not high level business functionalities directly usable by the end user at the business level, but they represent technical services that could be easily reused in a Service Oriented context. Technical services providing parallelism functionalities and hiding the complexity of grid parallelism management details, could be used to build higher level functional services that may internally be composed of several technical services. The advantages of this approach are the separation of concerns and the definition of a common and standard API able to introduce a loosely coupling dependence between the involved parties.



**Figure 1.1. An high level view of a SOA / GRID infrastructure**

The picture above aims at giving a view of a possible use of ProActive Parallel Services in a Service Oriented Architecture. The picture shows the main components: the end user can access to some higher level functionalities using his specific application. This end user application represents the entry point to the business processes of the organization, or to some of them. When the user triggers the execution of a business process, its execution is started and controlled by the Business Process Engine (typically a BPEL Engine). A process that is composed using some ProActive Parallel Services, invokes a Parallel Service at some point to leverage Grid capabilities. The Service creates ProActive jobs and submits them to the ProActive Scheduler running on the Grid Infrastructure. The Scheduler interacts with the ProActive Resource Manager to retrieve Grid resources where to schedule jobs on. When the Grid computation is

finished, the result is returned back to the business process that can continue its elaboration and can give back the business result to the end user.

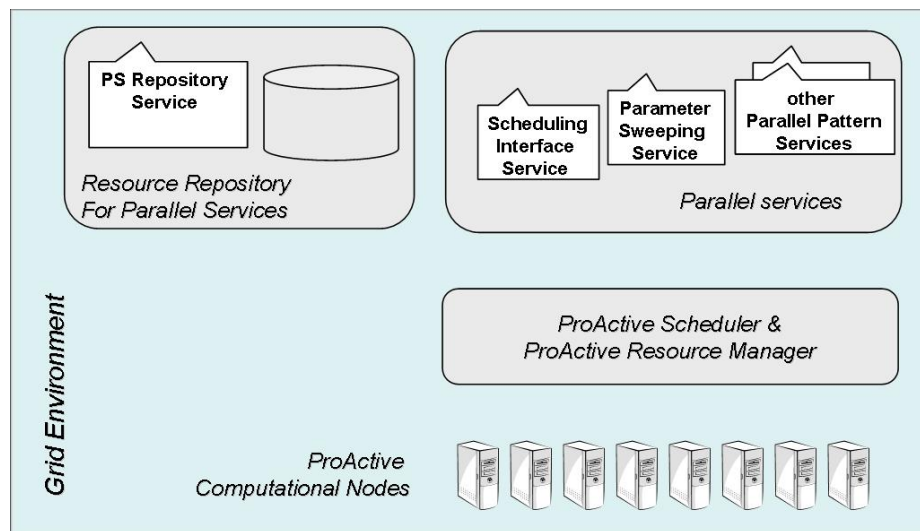
### 1.3. The Parallel Services set

In ProActive Parallel Services, there are two kinds of services:

- Scheduler Interface service: A Web Service exposing main functionalities of the ProActive Scheduler
- Parallel Pattern Services (finite set): Typical parallel computing patterns exposed as Web Services. The client has to specify its application specific code in order to execute it according to the desired pattern. Typical parallel computing patterns could be Parameter Sweeping, Divide And Conquer, etc.

All of these Parallel Services use a support service that is a repository service, named *PSRepository*, used to store all the user resources needed to execute a Parallel Service.

Following an high level view of the whole set of the services we are speaking about.



**Figure 1.2. Parallel Services overview**

The parallel services represent the access point to the Grid environment via the ProActive Scheduler/Resource Manager installation. The repository service provides the storage management of the user resources (like archives of execution logic units, data and job resources) needed by the parallel service execution. In this way, the external user resources not accessible from inside the Grid infrastructure can be transferred on the Grid side; the resource transfer happens in a phase distinct from the call and execution of the service instance that will use it, so that an uploaded resource can be reused in more than one service execution and the service call is lighter than the case of a big data transfer during the service call.

# Chapter 2. Functional Description of Parallel Services

## 2.1. PSRepository: a repository for Parallel Services user resources

### 2.1.1. Overview

As briefly introduced in [Section 1.1, "Introduction"](#), the PSRepository is a repository where a user can store the resources needed to execute a Parallel Service. Basically, the idea is that the organization that manages the Grid Infrastructure could provide to external users the possibility to access the Grid computational power using standard Web Services. A call to a Parallel Service will trigger on the Grid side the execution of a job running the user's logic on the user's data. Since the user is supposed to be outside of the Grid environment, the user's logic and data have to be transferred on the Grid side. One possibility could be to transfer from the user side to the Grid side all these resources during the call to a Parallel Service. This solution has two drawbacks: the call to the Parallel Service could be quite long if the resource is heavy and a new execution of the same job but running on different input for example, will require a new non-needed transfer. That is why a Repository service has been introduced. Its goal is to provide to the user the possibility to store on the Grid side the resources needed to execute his job. These resources have to be archived in one file, that will be transferred as a binary file calling the PSRepository Web Service. Once done, the user can request the execution of a Parallel Service referring the archive to use to perform the job.

### 2.1.2. The PSRepository functionalities

This section gives a view on the functionalities provided by the PSRepository Web Service.

Basically the service offers functionalities to store, list and retrieve archives stored in the repository. Each archive is a compressed file containing resources to execute a Parallel Service. Since the resources needed to execute different Parallel Services could differ, each archive has to be characterized with a specific type, depending on the Parallel Service it should be used for, in order to be validated for that use. These functionalities are represented by the following operations:

**storeArchive** - This operation stores in the repository the archive provided by the user.

- **Input:** the archive. An 'archive' contains the binary data of the zip archive file and also some meta information contained in an 'archiveInfo', such as a 'name', a textual 'description' and a 'type'.
- **Output:** the archive reference. It's a string representing the unique identifier of the archive in the repository.
- **Fault:** a storeArchive\_faultMsg if the archive is not a valid archive or if an error occurs during the process execution.

**listArchivesByOwner** - This operation gives to the user the possibility to retrieve the list of the archives owned by a specified user.

- **Input:** the user name of the owner user. In the current version there are no special policy to restrict the access to information about the archives existing in the repository.
- **Output:** a list of 'archiveProperties'. An 'archiveProperties' is a set of information related to an archive stored in the repository, but it does not contain the binary data of the archive. The information contained are: the archiveInfo (such as its name, its textual description and its type), the user name of the archive's owner and the archive's reference. The list is empty if no existing archive is owned by the given user.
- **Fault:** a listArchives\_faultMsg if an error occurs during the process execution.

**listArchivesByType** - This operation gives to the user the possibility to retrieve the list of the archives of a specified type.

- **Input:** the archive type. It is a string representing one of the possible archive type existing in the repository. (In the current version there are two possible types: 'JOB' and 'PARAMETER\_SWEEPING')

- **Output:** a list of 'archiveProperties'. An 'archiveProperties' is a set of information related to an archive stored in the repository, but it does not contain the binary data of the archive. The information contained are: the 'archiveInfo' (such as its name, its textual description and its type), the user name of the owner and the reference of the archive.
- **Fault:** a listArchives\_faultMs if an error occurs during the process execution.

**getArchive** - This operation allows a user to retrieve an archive having a given archive reference.

- **Input:** the reference of the archive.
- **Output:** the owner name of the archive (the current access policy lets a user retrieve only an archive owned by himself); the archive stored in the repository whose reference is the given one. The archive contains the binary data and the archiveInfo meta information.
- **Fault:** a getArchive\_faultMsg if an error occurs during the process execution or if the given archive reference does not exist in the repository.

Following the UML class diagram of the PSRepository Web Service interface to give a more detailed view on the used types (the fault messages, that have not a really relevant structure, are not represented to keep the diagram simpler):

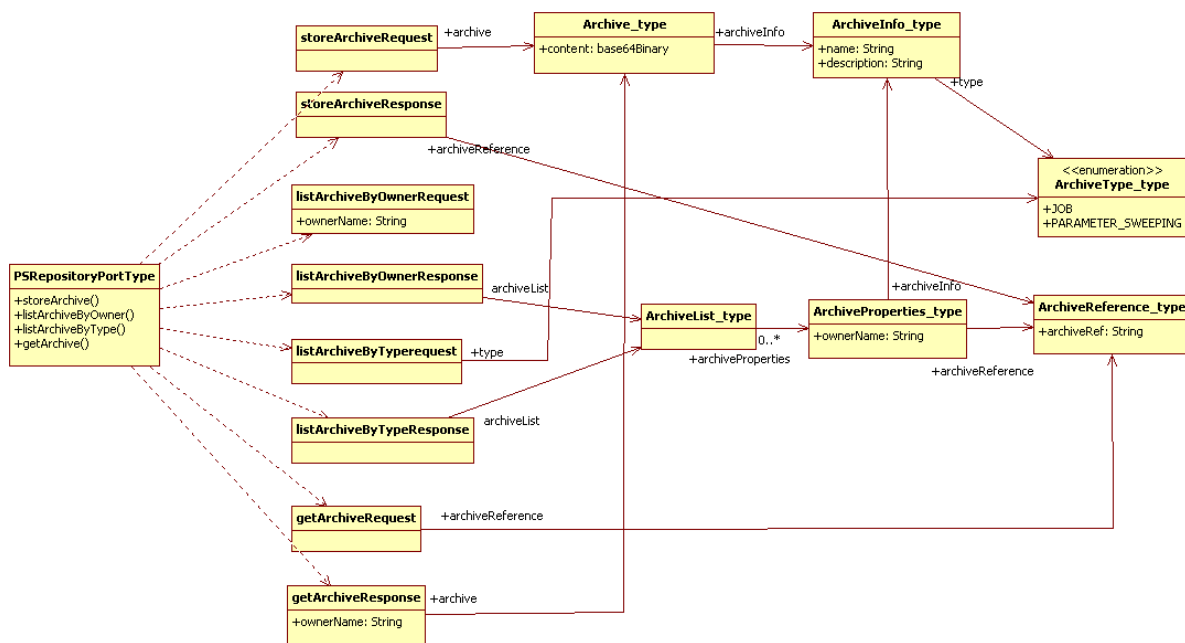


Figure 2.1. PSRepository Web Service interface

## 2.2. SchedulerInterface: a Web Service interface for ProActive Scheduler

### 2.2.1. Overview

This service is basically a web service exposing the main functionalities of the ProActive Scheduler. Even if the ProActive Scheduler has an easy to use API for a Java client, its interface would be too complex for a Web Service based client if it was exposed 'as is', using a code-first approach. So this service defines a simplified ProActive Scheduler Web Service interface. The current version of the Scheduler Interface exposes only non administrative functionalities, because actually it is assumed that the ProActive Scheduler is administered only from inside its administrative domain. Anyway a Web Service access to the ProActive Scheduler administration interface could be useful too, so its introduction could be considered for a further version.

### 2.2.2. The SchedulerInterface functionalities

This section gives a view on the functionalities provided by the SchedulerInterface Web Service.

As already said, the PSRepository service is used to store on the Grid side the resources needed to execute a Parallel Service. In the case of the SchedulerInterface service, an archive stored in the repository is used to launch the execution of a job, where the job is defined following the ProActive Scheduler's job description specifications (see the [ProActive Scheduler documentation](#)<sup>1</sup> for more details). The idea is to have the same kind of job descriptor to define a job, even if some limitations have to be taken into account, because of the loosely coupling between the user environment and the Grid environment where the Scheduler is supposed to run on. In order to reach this goal, the user has to provide his job descriptor as well as all the resources needed to execute the job as defined in the descriptor. All these resources are provided in an archive stored in the repository and having the 'JOB' type.

A valid *JOB* type archive has to respect a specific format. It should contain the following resources:

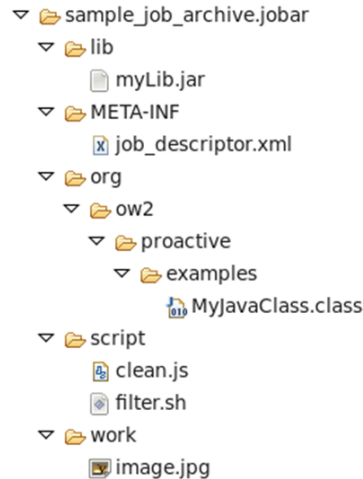
- **/META-INF** - (mandatory) - directory
- **/META-INF/jobdescriptor.xml** - (mandatory) - the job descriptor file (its file name must be 'jobdescriptor.xml').
- **/package/subpackage/classes.class** - (optional) - the user classes eventually needed by the job. The package directory is contained in the archive root.
- **lib** - (optional) - directory containing java libraries (.jar files) eventually needed by the job
- **script** - (optional) - directory containing user script files eventually needed by the job.
- **work** - (optional) - directory that represents the working directory of the job, if needed (it's the only resource where a running job has writing rights on)
- **/other-user-directories** - (optional) - user directories containing resources that won't be modified during job execution, if needed (obviously, these directories cannot be named as any of the other well known directories above).

The *jobdescriptor.xml* file is a standard ProActive xml job descriptor, but the following additional conventions must be followed in this case.

- Any kind of path relative to the user environment cannot be defined in the jobdescriptor.xml. The assumption is that the user's address space and the Scheduler server's address space are totally distinct. That is why all the resources that will be used in the job have to be included in the job archive.
- Script files used in the job descriptor have to be put in the **script** dir and referred in the descriptor only with their relative path to the **script** directory. So in the job descriptor, in 'script/file@path' and in 'staticCommand@value', we have the relative path of the used script, contained in the 'script' directory in the job archive.
- Any resource that will be used but will not be modified during the job execution can be put in a user defined directory in the job archive (as said above speaking about 'other-user-directories'). These resources must be referred in the jobdescriptor.xml using a well known variable `${JOB_ARCHIVE}`. For example, if the archive includes a user defined directory named 'mydir' containing an input file 'myfile.dat' referred somewhere in the job descriptor, the file has to be referred in the job descriptor as `'${JOB_ARCHIVE}/mydir/myfile.dat'`.
- Any resource that can be modified during the execution has to be included in the well known directory named 'work', and referred in the job descriptor using the variable `${WORK}`. For example, if the archive contains '/work/mydir/my-modifiable-file.dat', the job descriptor has to refer to this file as `'${WORK}/mydir/my-modifiable-file.dat'`.
- The JobClasspath parameter should not be defined. This parameter is equivalent to the Java Classpath and represents the classpath used to execute the Java tasks defined in the job. The job archive must include all the java libraries in the lib directory and all the user classes in the archive root; these resources represent the JobClasspath (it is implicitly defined) and if a JobClasspath parameter is defined by the user in the jobdescriptor.xml file, it will be ignored.
- The optional parameter 'logFile' cannot be defined.

Let's give an example of job archive in order to well understand its structure.

<sup>1</sup> [http://proactive.inria.fr/release-doc/Scheduler/multiple\\_html/user\\_manual.html#Create\\_job\\_xml](http://proactive.inria.fr/release-doc/Scheduler/multiple_html/user_manual.html#Create_job_xml)



**Figure 2.2. Job archive structure**

Following, we can see the content of the META-INF/job\_descriptor.xml file, that refers the resources contained in the job archive.

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns="urn:proactive:jobdescriptor:3.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:jobdescriptor:3.0 ../../src/scheduler/src/org/ow2/proactive/scheduler/
common/xml/schemas/jobdescriptor/3.0/schedulerjob.xsd"
  name="sample_job" priority="low" projectName="myProject"
  cancelJobOnError="true">
  <description>
    sample job to show the conventions to use in a job archive
  </description>
  <variables>
    <!-- this variable is used as default input of some tasks.
    Once the archive is stored, the job could be submitted with new
    input, giving a new value for the variable -->
    <variable name="STRING_1" value="myDefault" />
  </variables>

  <taskFlow>
    <!--
    A native task, with a pre-script.
    The scripts are contained in the 'script' dir.
    The scripts need a working directory: the 'work' dir
    will be used.
    -->
    <task name="filter">
      <pre>
        <script>
          <file path="clean.js">
            <arguments>
              <argument value="{WORK}" />
            </arguments>
          </file>
        </script>
      </pre>
    </nativeExecutable>
```

```

    <staticCommand value="filter.sh">
      <arguments>
        <argument value="{WORK}" />
      </arguments>
    </staticCommand>
  </nativeExecutable>
</task>

<task name="analyse" maxNumberOfExecution="2">
  <!--
    A java task: the class is contained in the archive and it
    needs the jar contained in the lib directory.
  -->
  <javaExecutable class="org.ow2.proactive.examples.MyJavaClass">
    <parameters>
      <parameter name="dirToAnalyse" value="{WORK}" />
      <parameter name="stringToPrint" value="{STRING_1}" />
    </parameters>
  </javaExecutable>
</task>

</taskFlow>
</job>

```

The functionalities provided by this Web Service are represented by the following operations:

**submitJobByRef** - This operation submits a job to the ProActive Scheduler where the job description and the needed resources to execute the job are included in a valid JOB type archive previously stored in the repository. The user can specify in the call some inputs that he would like to provide at the moment of the call. The inputs for a job are variables previously defined in the job description. In the job description, the user can define the variables used as inputs for the tasks. These variables, or some of them, can (nevertheless) be overridden during the call.

- **Input:** a job reference that is a reference to an existing JOB type archive in the repository and owned by the requiring user; some job inputs that consist of a set job Variables that will override existing Variables with the same name in the job description.
- **Output:** the job identifier.
- **Fault:** an invalidArchiveReference\_faultMsg if the archive reference is not a valid one; jobCreation\_faultMsg, if the job creation starting from the job descriptor and the provided resources in the archive fails; submitJob\_faultMsg if an error occurs during the process execution.

**getJobState** - This operation gives to the user the information about the state of a job identified by a given job identifier.

- **Input:** the job identifier
- **Output:** some information about the job, such as the job name, the owner, the job priority, the total number of tasks, the time of submission, the started time, etc.; information about the job's state, such as the status of the job (RUNNING, PENDING, ...) and the number of pending tasks, the number of running tasks and the number of finished tasks.
- **Fault:** an invalidJobReference\_faultMsg if the job reference is not a valid job id or if any existing job is identified by this reference; a getJobState\_faultMsg if an error occurs during the process execution.

**getJobResult** - This operation gives to the user the possibility to retrieve the result of a job identified by a given job identifier.

- **Input:** the job identifier; a boolean value saying if the user wants to retrieve only the precious results defined in the job descriptor, or all the results.

- **Output:** the job result includes the job's information, the job's state, a boolean saying if the job returned an exception, and eventually the list of all the precious task results, and eventually the list of all the partial task results too. Each task result specifies the task identifier, the task name, a boolean saying if the task returned an exception, eventually an exception message, eventually a result value and the output/error logs produced during the task execution.
- **Fault:** an `invalidJobReference_faultMsg` if the job reference is not a valid job id or if any existing job is identified by this reference; a `getJobResult_faultMsg` if an error occurs during the process execution.

**changeJobPriority** - This operation gives to the user the possibility to change the priority of a job identified by a given job identifier.

- **Input:**
  - the job identifier;
  - the priority to set to the specified job.
- **Output:** the new priority of the job.
- **Fault:**
  - `ChangeJobPriority_faultMsg` this message fault is built **IF** a well known exception occurs on the Scheduler side. *NOTICE: This fault message is **NOT** built if the job identifier is null or not valid or doesn't identify any job in the Scheduler's list of jobs.*
  - `InvalidJobReference_faultMsg`, this message fault is built **IF** it is impossible to identify the job to which we have to change the job priority. *NOTICE: this fault message is built **IF** the job identifier is empty (i.e., a string like "" or " ") or is a not valid string (in the actual implementation of the ProActive Scheduler, a valid identifier is a positive integer, except zero, represented as a string). This fault message is built **EVEN IF** a valid string is passed as job reference, but any job corresponding to that identifier exists in the list of jobs managed by the Scheduler.*
  - `InvalidJobPriority_faultMsg` this fault message is built **IF** the priority to set to the specified job is not admissible (e.g., it is a valid value, given the WSDL definition, but it is not acceptable for the ProActive Scheduler). *NOTICE: in the WSDL the priority is a restricted type (i.e., it can assume only a predefined set of values) so the client will get an **AXIS fault** message if he specifies a priority that does not belong to the set of admissible values. So the `InvalidJobPriority_faultMsg` should never arise.*
  - `Fault_faultMsg` this fault message is built **IF** an unpredictable exception occurs on the server side.

**getOutput** - This operation gives to the user the possibility to retrieve the output of a job identified by a given job identifier.

- **Input:** the job identifier.
- **Output:** a list in which each element is the output of a task of the job.
- **Fault:**
  - `GetOutput_faultMsg`, this fault message is built **IF** the output of the job does not exist. This fault message is built **EVEN IF** a well known exception occurs on the Scheduler side. *NOTICE: This fault message is **NOT** built if the job identifier is null or not valid or doesn't identify any job in the Scheduler's list of jobs.*
  - `InvalidJobReference_faultMsg`, this message fault is built **IF** it is impossible to identify the job whose output is required. *NOTICE: this fault message is built **IF** the job identifier is empty (i.e., a string like "" or " ") or is a not valid string (in the actual implementation of the ProActive Scheduler, a valid identifier is a positive integer, except zero, represented as a string). This fault message is built **EVEN IF** a valid string is passed as job reference, but any job corresponding to that identifier exists in the list of jobs managed by the Scheduler.*
  - `Fault_faultMsg` this fault message is built **IF** an unpredictable exception occurs on the server side.

**getTaskOutput** - This operation gives to the user the possibility to retrieve the output of a task identified by the job identifier, that identifies the job that owns the task, and the string that represents the name of the task.

- **Input:**
  - the job identifier of the job that owns the task whose output is desired;
  - the name of the task whose output is desired.
- **Output:**

- the identifier of the task, that includes the identifier of the job that owns the task and the name of the task;
- the string representing the output of the task.
- **Fault:**
  - GetTaskOutput\_faultMsg this fault message is built **IF** the output of the task does not exist. This fault message is built **EVEN IF** a well known exception occurs on the Scheduler side. *NOTICE: This fault message is NOT built if the job identifier is null or not valid or doesn't identify any job in the Scheduler's list of jobs.*
  - InvalidJobReference\_faultMsg, this message fault is built **IF** it is impossible to identify the job the referred task is part of. *NOTICE: this fault message is built IF the job identifier is empty (i.e., a string like "" or " ") or is a not valid string (in the actual implementation of the ProActive Scheduler, a valid identifier is a positive integer, except zero, represented as a string). This fault message is built EVEN IF a valid string is passed as job reference, but any job corresponding to that identifier exists in the list of jobs managed by the Scheduler.*
  - InvalidTaskReference\_faultMsg this fault message is built **IF** it is impossible to identify the task whose output has to be retrieved. *NOTICE: this fault message is built IF the specified task name string is empty. This fault message is built EVEN IF the Scheduler cannot identify the task (i.e., it is unable to find the task in the list of tasks that a job owns).*
  - Fault\_faultMsg this fault message is built **IF** an unpredictable exception occurs on the server side.

**getTaskResult** - This operation gives to the user the possibility to retrieve the result of a task identified by the job identifier, that identifies the job that owns the task, and the string that represents the name of the task.

- **Input:**
  - the job identifier of the job that owns the task whose output is desired;
  - the name of the task whose output is desired.
- **Output:** a structured information representing the result of the task execution. It includes:
  - the identifier of the task;
  - information that tell us if the task has thrown an exception during its execution;
  - the value of the task result;
  - the exception message;
  - the logs of the task, such as the standard and the error logs.
- **Fault:**
  - GetTaskResult\_faultMsg this fault message is built **IF** the result of the task does not exist. This fault message is built **EVEN IF** a well known exception occurs on the Scheduler side. *NOTICE: This fault message is NOT built if the job identifier is null or not valid or doesn't identify any job in the Scheduler's list of jobs.*
  - InvalidJobReference\_faultMsg, this message fault is built **IF** it is impossible to identify the job the referred task is part of. *NOTICE: this fault message is built IF the job identifier is empty (i.e., a string like "" or " ") or is a not valid string (in the actual implementation of the ProActive Scheduler, a valid identifier is a positive integer, except zero, represented as a string). This fault message is built EVEN IF a valid string is passed as job reference, but any job corresponding to that identifier exists in the list of jobs managed by the Scheduler.*
  - InvalidTaskReference\_faultMsg this fault message is built **IF** it is impossible to identify the task whose result has to be retrieved. *NOTICE: this fault message is built IF the specified task name string is empty. This fault message is built EVEN IF the Scheduler cannot identify the task (i.e., it is unable to find the task in the list of tasks that a job owns).*
  - Fault\_faultMsg this fault message is built **IF** an unpredictable exception occurs on the server side.

**killJob** - This operation gives to the user the possibility to kill a job identified by a given job identifier

- **Input:** the identifier of the job to kill.
- **Output:**
  - the informations about the job such as: the job identifier, the name of the job, the owner of the job, the priority of the job, the time at which the job was submitted, the time at which the job has started its execution, the total number of tasks composing the job;

- the informations about the state of the job such as: the status of the job and the list of states of all the tasks the job owns. Each element of this list contains the identifier of the task, its name, its status, the host the task is executed on, the duration of its execution, the start and end time, the number of executions left, the number of executions on failure left and the number of executions on failure.
- **Fault:**
  - KillJob\_faultMsg this fault message is built **IF** the job cannot be killed. This fault message is built **EVEN IF** a well known exception occurs on the Scheduler side. *NOTICE: This fault message is **NOT** built if the job identifier is null or not valid or doesn't identify any job in the Scheduler's list of jobs.*
  - InvalidJobReference\_faultMsg, this message fault is built **IF** it is impossible to identify the referred job. *NOTICE: this fault message is built **IF** the job identifier is empty (i.e., a string like "" or " ") or is a not valid string (in the actual implementation of the ProActive Scheduler, a valid identifier is a positive integer, except zero, represented as a string). This fault message is built **EVEN IF** a valid string is passed as job reference, but any job corresponding to that identifier exists in the list of jobs managed by the Scheduler.*
  - Fault\_faultMsg this fault message is built **IF** an unpredictable exception occurs on the server side.

**listJobs** - This operation gives to the user the possibility to retrieve a list of jobs the Scheduler manages. This list is made up of a contiguous set of jobs.

- **Input:**
  - the string representing the name of the owner a job must belong to in order to be included in the result list;
  - the status a job must have in order to be included in the result list. The status can be one of the following: PENDING, RUNNING, STALLED, FINISHED, PAUSED, CANCELED, FAILED, KILLED;
  - the maximum number of jobs to include in the result list;
  - the index of the first job that will be included in the result list. This index represents the position at which the job is stored in the list of jobs the Scheduler manages;
  - an information that says if the index (i.e., the integer that represents the position at which the job is stored in the list of jobs the Scheduler manages) is relative to the beginning or the end of the list of jobs managed by the Scheduler.
- **Output:**
  - the informations about the job such as: the job identifier, the name of the job, the owner of the job, the priority of the job, the time at which the job was submitted, the time at which the job has started its execution, the total number of tasks composing the job;
  - the informations about the state of the job such as: the status of the job and the list of states of all the tasks the job owns. Each element of this list contains the identifier of the task, its name, its status, the host the task is executed on, the duration of its execution, the start and end time, the number of executions left, the number of executions on failure left and the number of executions on failure.
- **Fault:**
  - ListJobs\_faultMsg this fault message is built **IF** a well known Scheduler exception occurs.
  - Fault\_faultMsg this fault message is built **IF** an unpredictable exception occurs on the server side.

**pauseJob** - This operation gives to the user the possibility to pause a job identified by a given job identifier.

- **Input:** the job identifier
- **Output:**
  - the informations about the job such as: the job identifier, the name of the job, the owner of the job, the priority of the job, the time at which the job was submitted, the time at which the job has started its execution, the total number of tasks composing the job;
  - the informations about the state of the job such as: the status of the job and the list of states of all the tasks the job owns. Each element of this list contains the identifier of the task, its name, its status, the host the task is executed on, the duration of its execution, the start and end time, the number of executions left, the number of executions on failure left and the number of executions on failure.
- **Fault:**

- **PauseJob\_faultMsg** this fault message is built **IF** the job cannot be paused. This fault message is built **EVEN IF** a well known exception occurs on the Scheduler side. *NOTICE: This fault message is **NOT** built if the job identifier is null or not valid or doesn't identify any job in the Scheduler's list of jobs.*
- **InvalidJobReference\_faultMsg**, this message fault is built **IF** it is impossible to identify the referred job. *NOTICE: this fault message is built **IF** the job identifier is empty (i.e., a string like "" or " ") or is a not valid string (in the actual implementation of the ProActive Scheduler, a valid identifier is a positive integer, except zero, represented as a string). This fault message is built **EVEN IF** a valid string is passed as job reference, but any job corresponding to that identifier exists in the list of jobs managed by the Scheduler.*
- **Fault\_faultMsg** this fault message is built **IF** an unpredictable exception occurs on the server side.

**removeJob** - This operation gives to the user the possibility to remove a job identified by a given job identifier.

- **Input:** the job identifier
- **Output:** the job identifier of the removed job
- **Fault:**
  - **RemoveJob\_faultMsg** this fault message is built **IF** the job cannot be removed. This fault message is built **EVEN IF** a well known exception occurs on the Scheduler side. *NOTICE: This fault message is **NOT** built if the job identifier is null or not valid or doesn't identify any job in the Scheduler's list of jobs.*
  - **InvalidJobReference\_faultMsg**, this message fault is built **IF** it is impossible to identify the referred job. *NOTICE: this fault message is built **IF** the job identifier is empty (i.e., a string like "" or " ") or is a not valid string (in the actual implementation of the ProActive Scheduler, a valid identifier is a positive integer, except zero, represented as a string). This fault message is built **EVEN IF** a valid string is passed as job reference, but any job corresponding to that identifier exists in the list of jobs managed by the Scheduler.*
  - **Fault\_faultMsg** this fault message is built **IF** an unpredictable exception occurs on the server side.

**resumeJob** - This operation gives to the user the possibility to resume the execution of a job identified by a given job identifier.

- **Input:** the job identifier of the job to resume
- **Output:**
  - the informations about the job such as: the job identifier, the name of the job, the owner of the job, the priority of the job, the time at which the job was submitted, the time at which the job has started its execution, the total number of tasks composing the job;
  - the informations about the state of the job such as: the status of the job and the list of states of all the tasks belonging to the job. Each element of this list contains the identifier of the task, its name, its status, the host the task is executed on, the duration of its execution, the start and end time, the number of executions left, the number of executions on failure left and the number of executions on failure.
- **Fault:**
  - **ResumeJob\_faultMsg** this fault message is built **IF** the job cannot be resumed. This fault message is built **EVEN IF** a well known exception occurs on the Scheduler side. *NOTICE: This fault message is **NOT** built if the job identifier is null or not valid or doesn't identify any job in the Scheduler's list of jobs.*
  - **InvalidJobReference\_faultMsg**, this message fault is built **IF** it is impossible to identify the referred job. *NOTICE: this fault message is built **IF** the job identifier is empty (i.e., a string like "" or " ") or is a not valid string (in the actual implementation of the ProActive Scheduler, a valid identifier is a positive integer, except zero, represented as a string). This fault message is built **EVEN IF** a valid string is passed as job reference, but any job corresponding to that identifier exists in the list of jobs managed by the Scheduler.*
  - **Fault\_faultMsg** this fault message is built **IF** an unpredictable exception occurs on the server side.

Following the UML class diagram of a subset of the SchedulerInterface Web Service interface entities, to give a view of the types used:

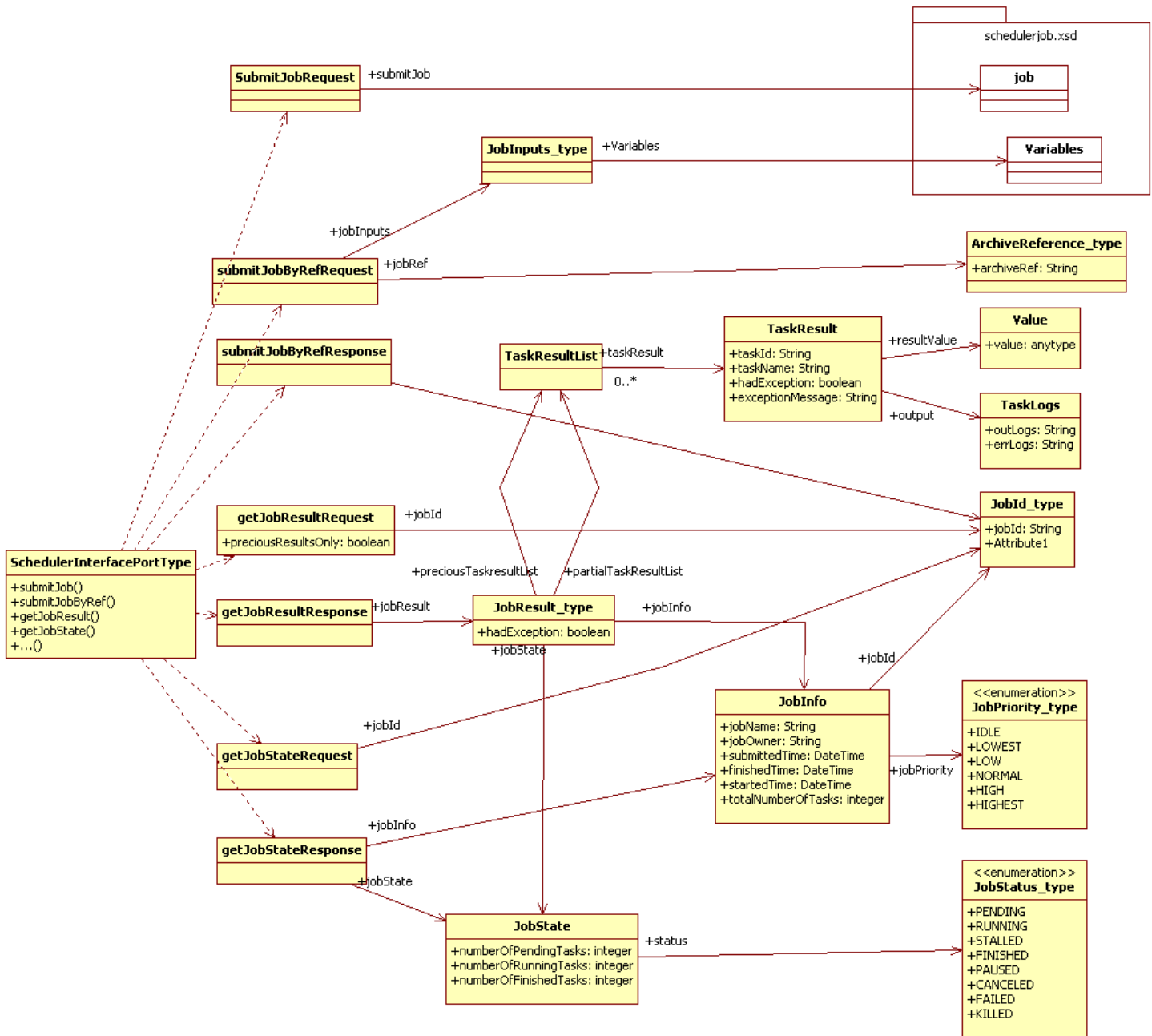


Figure 2.3. SchedulerInterface interface (partial view)

## 2.3. ParameterSweeping: a parallel computation pattern running jobs on the ProActive Scheduler.

### 2.3.1. Parallel Pattern Services

A Parallel Pattern Service is a typical parallel computing patterns exposed as Web Service. In this version of the software only this service is provided in this category of services. A user needing Grid computational power for a specific problem can use one or several Parallel Pattern Services to customize the behavior of each service with his business logic, according to the pattern that each service

implements. The customization of the service behavior is done extending the hotspots that are defined by each service according to the its specific pattern. The user can store in the repository his business logic to customize the pattern, as said above, and then he is able to call the desired parallel pattern service customized with his business code. The specific Parallel Pattern Service is in charge of building the job to run the user logic on the Grid nodes, according to the specific pattern. All the concerns relative to the parallelization and the distribution of the tasks on the Grid are up to the Parallel Pattern Service implementation.

### 2.3.2. Parameter Sweeping Overview

The Parameter Sweeping parallel pattern, is a simple pattern characterized by task replication where a specific logic must be executed in parallel over multiple values given as input. The idea is to sweep over all the given values of the input parameters and run the logic over each value, obtaining for each execution an output value. All the output values represent the whole output of the execution. The business logic is contained in a `PARAMETER_SWEEPING` type archive in the repository service. The archive contains, at the root level, the user packages and classes if any, and the 'lib' directory (located at the root level too) contains the user's Java libraries that are needed for the job execution.

### 2.3.3. Parameter Sweeping HotSpots - User Specific Implementation

In order to use the Parameter Sweeping Service, one should provide specific implementation for the two main operations:

- **sweep parameters operation** - defines the process that 'cuts' the input data into a set of 'slices'. Each slice is to be treated by a separate process.
- **treat data operation** - defines the process that 'treats' one slice of data.

The Parameter Sweeping Service proposes two hot spots to be extended in order to provide a concrete implementation of the above mentioned operations.

When implementing the hot spots, knowledge on the Parameter Sweeping Internal implementation nor on the Web Service's API are necessary. The only API one should use is the one defined by the interfaces and abstract classes which compose the hot spots:

- `ParamSweeperInterface` - interface which defines methods for the sweep parameters operation
- `AbstractParameterSweeper` - abstract class which implements the `ParamSweeperInterface`. Users should provide a concrete implementation of this class in order to extend the Sweeper hot spot
- `DataSliceTreater` - interface which defines methods for the treat data operation
- `AbstractDataSliceTreater` - abstract class which implements the `DataSliceTreater` interface. Users should provide a concrete implementation of this class in order to extend the Treater hot spot

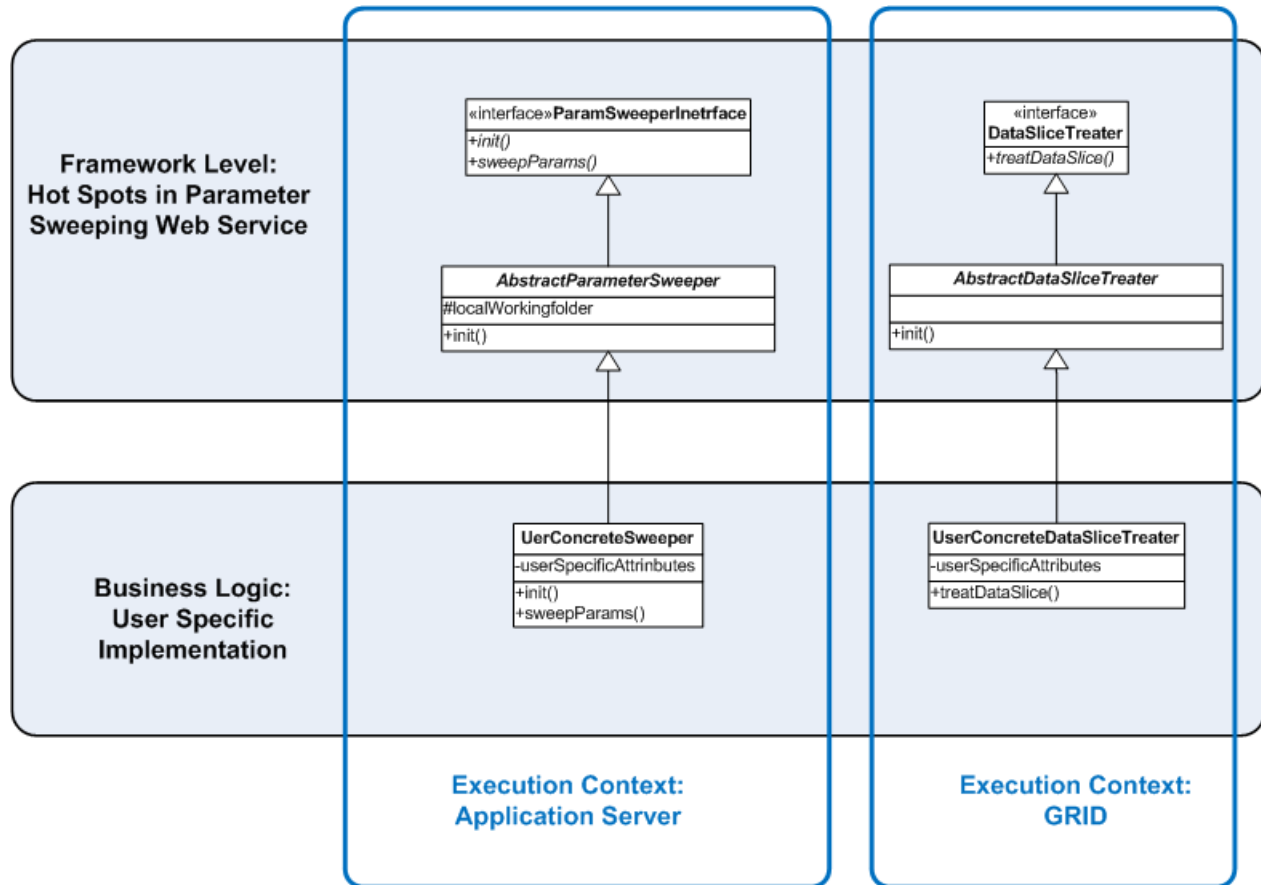


Figure 2.4. ParameterSweeping Hot Spots

### Concrete implementation of AbstractParameterSweeper

The `AbstractParameterSweeper` class implements the interface's method `init()`:

```
public boolean init(Map<String, Object> configuration)
```

The configuration map corresponds to the `configurationData` argument given by the user when calling the Parameter Sweeping Web Service. It should contain business logic specific data for the configuration of the Sweeper, before the sweeping operation. This information may be used in order to take decisions during the sweeping algorithm and/or when creating the data slices to be passed to the Treater.

In the abstract class, the value of the local attribute `localWorkingFolder` is set in the `init` method. It represents the absolute path of a folder where concrete implementations of the class should be stored. Therefore, when overriding this method, one should first call `super.init(configuration)`.

When extending `AbstractParameterSweeper`, one must implement the method `sweepParams` declared in the `ParamSweeperInterface`:

```
public List<Map<String, Serializable>> sweepParams(Map<String, Object> inputData) throws  
InvalidParametersException;
```

The `inputData` map represents the entries provided by the user, as `inputData` argument, when calling the Parameter Sweeping service. The keys and values contained in this map are specific to the business logic of the application.

If the sweeping process needs to create files that should be accessed by the `DataSliceTreater` implementation, these files must be put at the location "localWorkingFolder", declared and set by the abstract class. The Web Service uses ProActive Data Spaces in order to transfer these files to a remote node, when needed.

The output of this method is a list of maps. The content of the maps depends on the business logic specific to the user application. Each map will be passed (by the Parameter Sweeping Service, via ProActive Scheduler) as argument for the `treatDataSlice(...)` method defined by the Treater Hot Spot. Therefore, the number of tasks (processes to be executed on the remote nodes) is equal to the number of maps in the output.

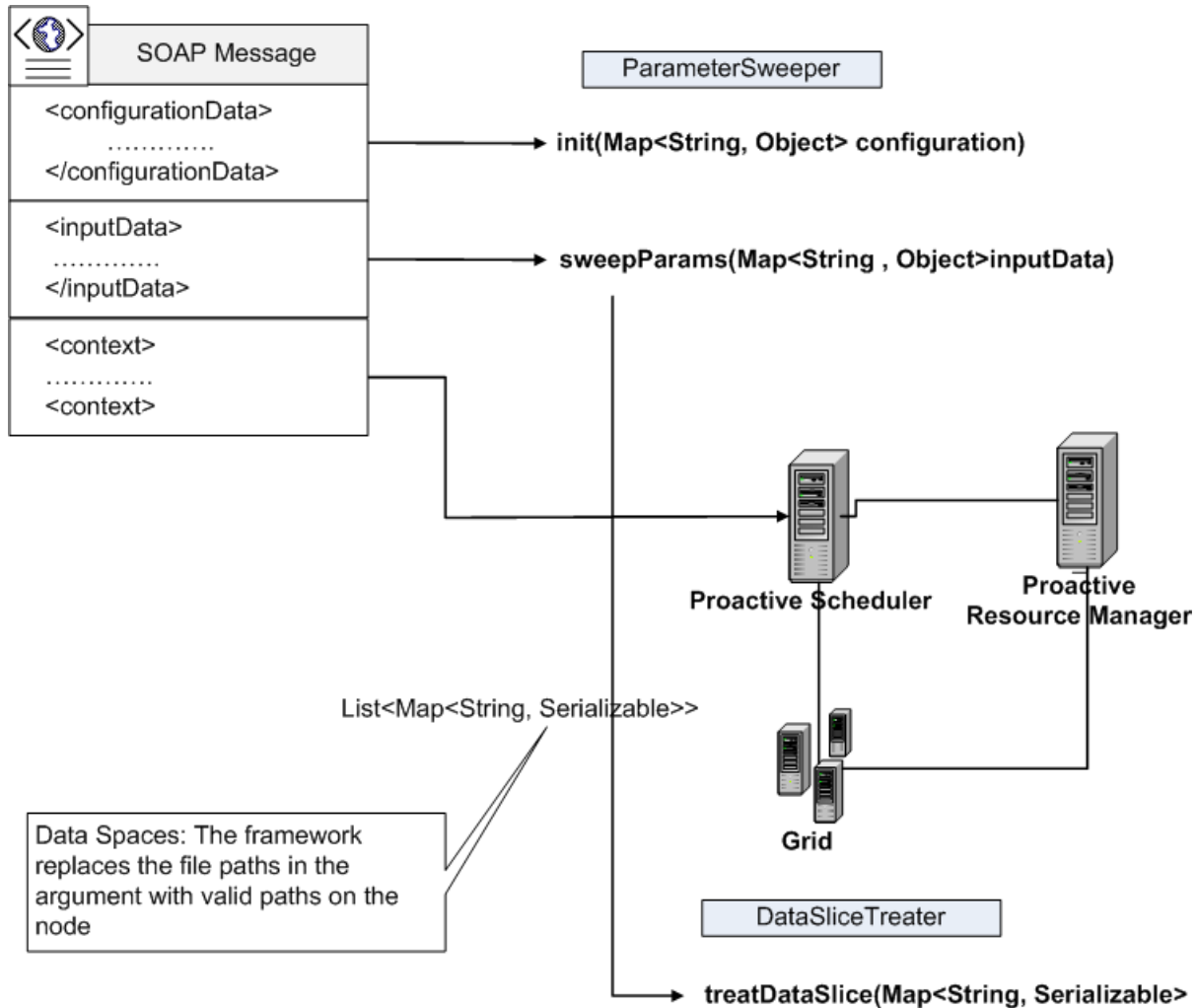
### Concrete implementation of `AbstractDataSliceTreater`

When extending the `AbstractDataSliceTreater` one must implement the method:

```
public abstract Serializable treatDataSlice(Map<String, Serializable> dataSlice) throws Exception;
```

As specified above, the `dataSlice` argument corresponds to one of the maps contained in the output of the `sweepParams(...)` method, of the treater. The fact that the execution of this method will take place on a working node is completely transparent to the developer. If one has to create files within this method, they have to be put at the location `outputFolderPath`, defined by the superclass as absolute path. This files will then be automatically managed by ProActive Data Spaces and copied to the Application Server when this method finishes. The output of this method could be the final result of the treatment.

Bellow, a figure representing the argument passing in the system:



**Figure 2.5. ParameterSweeping - passing arguments between different entities**

### 2.3.4. The ParameterSweeping Functionalities

This section gives a view on the functionalities provided by the ParameterSweeping Web Service.

The ParameterSweeping service presents in its WSDL definition three Port Types. The first Port Type, named 'ParameterSweepingPortType' includes the operation for a synchronous interaction between the service provider and the requester. The other two Port Types are used to realize an asynchronous interaction between the two parties. The first of these two port types, named 'AsynchParameterSweepingPortType' is implemented by the service provider, while the second one, named 'AsynchParameterSweepingCallBackPortType' has to be implemented by the service requester. It's important to put in evidence that even if several solutions can be used to support asynchronous interactions, and even if some of them satisfy this requirement at a lower level, this kind of solution seems to be the most accepted one. Besides, it follows the WS-I Basic Profile recommendation concerning this aspects. That is important in terms of interoperability and of reusability of the Parallel Pattern Services for external users that need services exposed by the Grid infrastructure.

The ParameterSweepingPortType presents the following operation:

#### runParamSweep

- **Input:** the archive reference to the PARAMETER\_SWEEPING type archive including the business code needed to customize the Parameter Sweeping pattern execution; the complete name of the two Java class contained in the archive and implementing the two

hot spots; some input data, and some configuration data, both represented as a map of keys-values: the input data are the effective input where to sweep over to obtain the data slice for each task, while the configuration data could be data needed to perform some initialization operations (as described above in [Section 2.3.2, “Parameter Sweeping Overview”](#)); the additional context information input, could contain information related to the client context and that will be associated to the resulting job. This kind of information could be used in different way, depending on the whole environment the service is installed in. For example, the association of information related to the client context could be useful to influence the scheduling policy (if the Scheduler instance has been customized with that); furthermore this information, that is associated with the requester and the resulting job, could be simply retrieved from both sides by an external monitor tool in order to perform a global monitoring of the whole environment and of the entities running on it.

- **Output:** the output is represented by a map of keys-values resulting from the treatment of all the tasks.
- **Fault:** an InvalidArchiveReferenceFault if the archive reference is not a valid one; a RunParamSweepFault if an error occurs during the process execution.

The AsynchParameterSweepingPortType presents the following operation:

#### initiateRunParamSweep

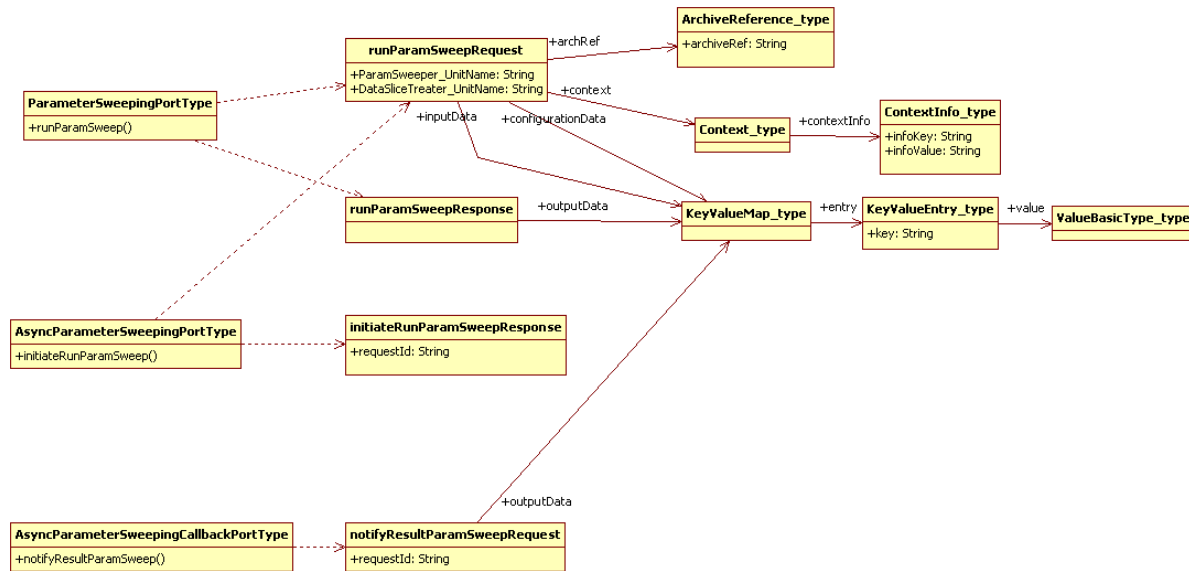
- **Input:** same input as the runParamSweep operation in the ParameterSweepingPortType.
- **Output:** a requestId representing the identifier of the submitted request. This value will be used during the notification of the result, when the service provider will call back the requester calling the operation defined in the 'AsynchParameterSweepingCallBackPortType' on the service end point listening on the requester side.
- **Fault:** an InvalidArchiveReferenceFault if the archive reference is not a valid one; a RunParamSweepFault if an error occurs during the process execution.
- **Additional information:** in the request, the requester has to provide some additional information to the provider, in order to let the provider call back him when the result will be ready. This information consists of the physical end-point where the requester exposes the service implementing the call back Port Type, and some correlation information to be able to associate the message content received from the service provider during the call back, with the correct client process in charge of the result collection (because the same requester can perform several calls to the same provider and so it can wait for several call back invocation). All of that has to be treat following the WS-Addressing standard, so that these information have to be added to the SOAP Header of the initiation and the call back request.

The AsynchParameterSweepingCallBackPortType, as already said, is the Port Type to be implemented by the requester. It presents the following operation:

#### notifyResultRunParamSweep

- **Input:** the requestId that has been produced as the result of the execution, represented by a map of keys-values resulting from the treatment of all the tasks.
- **Output:** no output message
- **Fault:** a NotifyResultFault if an error occurs during the process execution.
- **Additional information:** in this call back request, the provider, that is the caller of this operation, has to provide the correlation information that the requester has eventually put in the SOAP Header of the request in order to let him do the correlation between his request to the 'initiateRunParamSweep' operation and the request received on this call back operation.

The UML class diagram of the ParameterSweeping Web Service interface is exposed hereafter to give a more detailed view on the types used (the fault messages, that have not a really relevant structure, are not represented to keep the diagram simpler).



**Figure 2.6. ParameterSweeping interface**

## 2.4. Security requirements

All the Web Services described above are securized using the WS-Security standard specifications. The WS-SecurityPolicy implemented is the UsernameToken policy and HTTPS connection is required at transport level. The implementation is based on Rampart, the Axis2 module for WS-Security support.

# Chapter 3. Implementation details

## 3.1. Overview

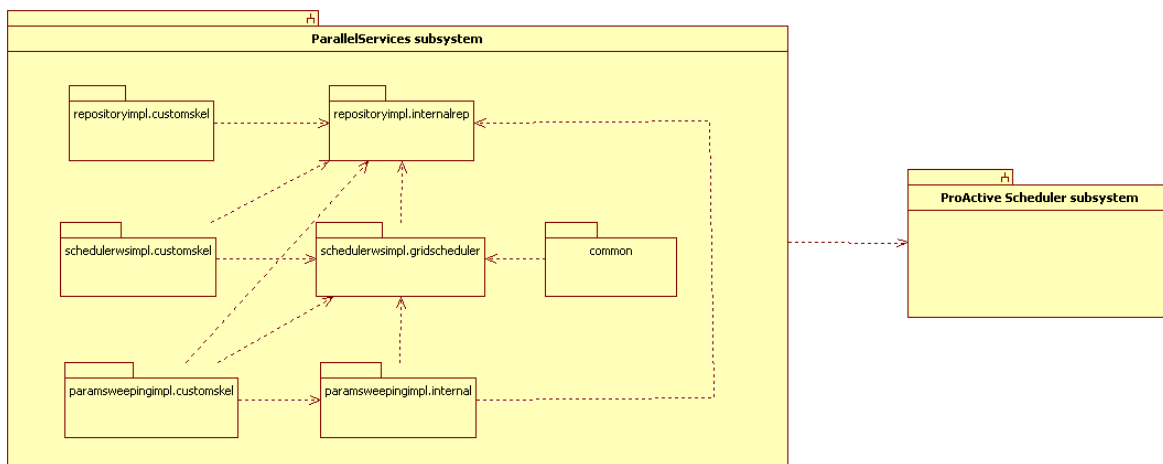
The Parallel Services implementation is based on the Apache Axis2 Web Services open source engine.

The development of all the services follows a WSDL first approach in order to exactly define all the details of the WSDL/XSD definitions, which represent the interface effectively seen by external requesters. Once defined these interfaces, the code auto-generated by Axis2 will manage marshalling/unmarshalling of SOAP messages, while some skeleton classes implement the business logic of the service. The following diagram shows a view of the main packages of the ParallelServices subsystem.



### Warning

In the whole chapter, all the package names are not reported entirely for a better visualization (all the reported package's complete names have the suffix `org.ow2.proactive.parallelservices.`).



**Figure 3.1. Parallel Services subsystem**



### Note

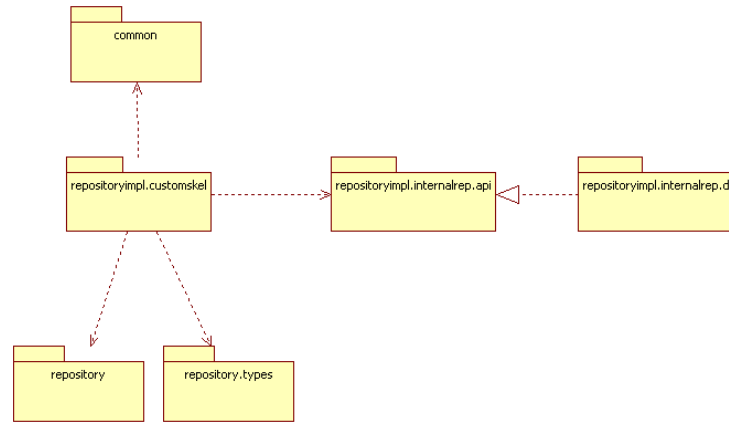
In the ParallelServices subsystem only the main packages are shown (the autogenerated packages are not reported in this diagram). For each service a more detailed view is given in the sections below.

The ParallelServices subsystem depends directly on the ProActive Scheduler.

For each service the package having the name suffix `.customskel` contains the skeleton classes and any other Axis2 specific class customized with user's logic to override the default Axis2 behavior. For each service, the only package that depends on the WSDL definition is the `.customskel` package. For the PSRepository service, the `repositoryimpl.internalrep` contains the core logic of the service. In the same way, `schedulerwsimpl.gridscheduler` represents the SchedulerInterface's core logic, as `paramsweepingimpl.internal` represents the ParameterSweeping's core logic. A `common` package is used by all the `.customskel` packages, even if these dependencies are not reported in the diagram to keep it lighter.

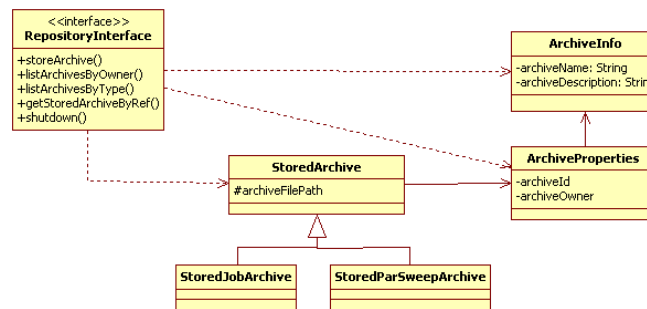
## 3.2. PSRepository implementation details

The PSRepository service implementation includes several main packages, as shown in the following diagram:



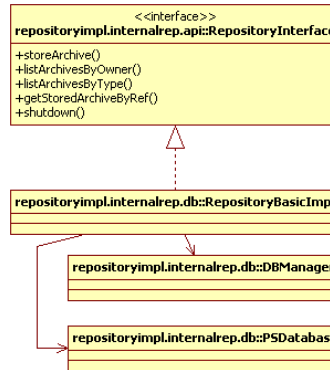
**Figure 3.2. PSRepository main packages**

As said above, the `repositoryimpl.customskel` package contains the main skeleton class used to define the business logic of the service and classes that override the default behavior of the service once deployed in the Axis2 container. It is the only package that depends on the Axis2 automatically generated classes, contained in the two packages `repository` and `repository.types`, that are needed to manage the marshalling/unmarshalling of SOAP messages. The PSRepository core implementation is contained in the `repositoryimpl.internalrep` package, which is divided into two sub-packages. The `repositoryimpl.internalrep.api` contains classes and interfaces that define a simple API designed to provide the functionalities needed by the PSRepository service.



**Figure 3.3. Repository Interface API**

The package `repositoryimpl.internalrep.db` contains a Data Base based implementation of that API.

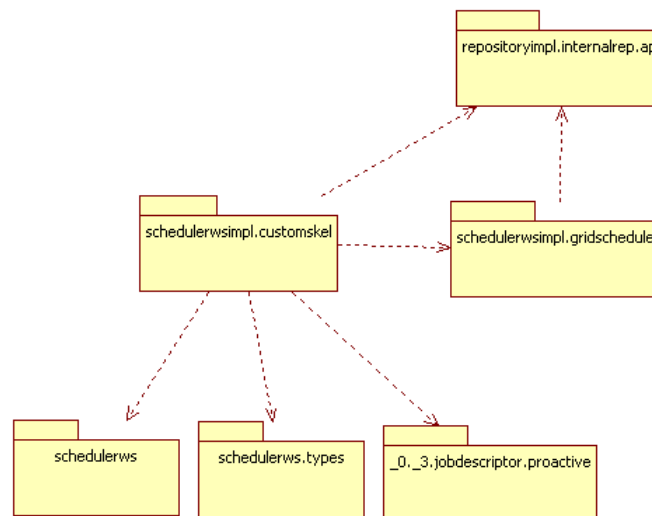


**Figure 3.4. Repository Interface data base implementation**

The current implementation stores in a Data Base all the meta information about stored archives, while the binary content of each archive is stored in a configured location on the file system. The Data Base access is based on JDBC. In the current release the open source relational database [Apache Derby](http://db.apache.org/derby/index.html)<sup>1</sup> is used as in memory Data Base Management System.

### 3.3. SchedulerInterface implementation details

The SchedulerInterface service implementation includes several main packages, as shown in the following diagram:



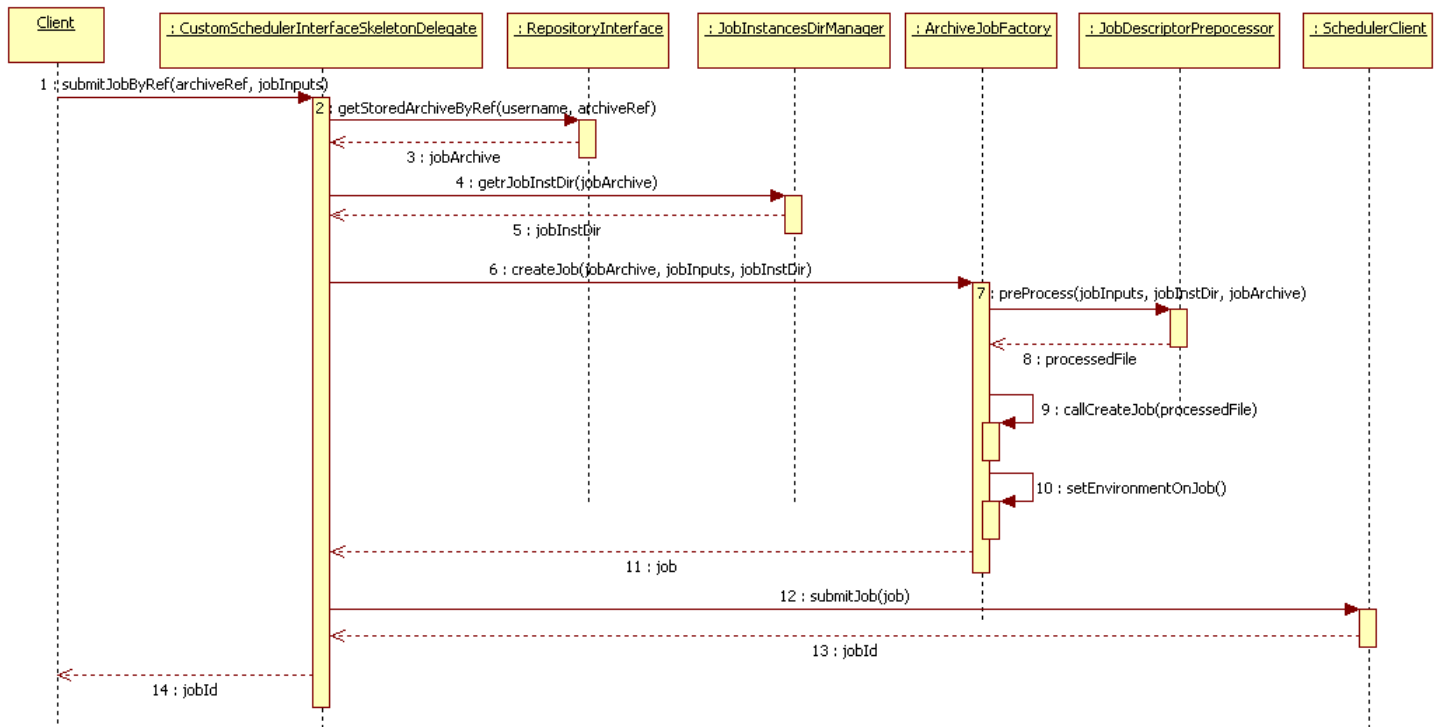
**Figure 3.5. SchedulerInterface main packages**

The `schedulerwsimpl.customskel` package contains the main skeleton class used to define the business logic of the service and classes that override the default behavior of the service once deployed in the Axis2 container. It is the only package that depends on the Axis2 automatically generated classes, contained in the packages `schedulerws`, `schedulerws.types` and `_0._3.jobdescriptor.proactive`, that are needed to manage the marshalling/unmarshalling of SOAP messages. This package depends on the package `repositoryimpl.internalrep.api`, that manages the archives stored in the repository and accessed by this service, and it depends also on the package `schedulerimpl.gridscheduler` that contains the main logic to access the ProActive grid scheduler. Basically the package `schedulerimpl.gridscheduler` contains the classes that access the ProActive Scheduler, adapting the behaviour

<sup>1</sup> <http://db.apache.org/derby/index.html>

to the requirements of the ParallelServices. There are classes that are in charge of the simple access to the ProActive Scheduler functions, classes that are in charge of the asynchronous behaviour of some services, and classes that are in charge of the management of the job submission starting from the archive stored in the repository via the PSRepository service

Let's give an idea of the interactions between main classes involved in the job submission via the a Job archive.

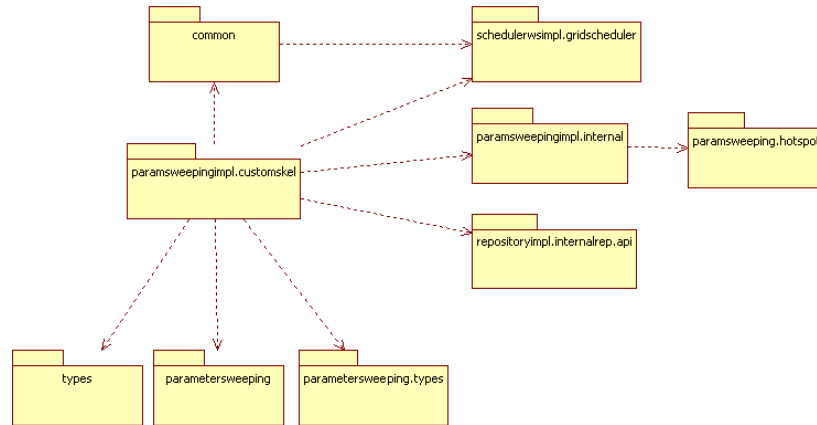


**Figure 3.6. Job Submission via archive**

In the picture reported above, we can see the sequence of the main steps followed by the implementation in order to submit a job to the ProActive Scheduler starting from an existing archive and giving new inputs, as described in the functional description of the SchedulerInterface. In the current implementation, once the client invokes the submitJobByRef operation on the SchedulerInterface service, the CustomSchedulerInterfaceSkeletonDelegate is invoked by Axis2 (step 1) and the implementation starts retrieving from the repository (via the RepositoryInterface) the archives previously stored with the archive reference archiveRef (step 2). Once retrieved the stored job archive, the JobInstanceDirManager is called (step 4). The JobInstanceDirManager is a manager of the directories that are dedicated to the execution of the jobs. As said in the functional description, for each job a working directory can be used by the user if needed, so this entity is in charge of instantiate that. The location for the working directory root is configured by the user in the schedconf.properties configuration file. At this point the ArchiveJobFactory can create the job starting from the job archive, the new job inputs given by the client, and the working directory dedicated to that instance (step 6). The job archive could contain several directories (for lib, scripts, etc.) and the path of these directories has to be referenced in the job descriptor file using some predefined variables. So a JobDescriptorPreprocessor is involved in the interaction at this point (see the step 7 in the diagram) in order to add in the job descriptor file the predefined variables that point to the referenced directories. In this step, the preprocessor is also in charge of update in the job descriptor, the value of the variables given as new input for the current job execution. At this point the job descriptor is ready to be created (step 9) and at this point the location of all the classes and lib files needed by the job are added to the job environment (step 10). Now the job is submitted to the Scheduler (step 12) and the job identifier is returned back to the client.

### 3.4. Parameter Sweeping implementation details

The packages included in the Parameter Sweeping Web Service implementation are shown in the figure bellow.



**Figure 3.7. Parameter Sweeping main packages**

As said above, the `paramsweepingimpl.customskel` package contains the main skeleton classes to define the business logic of the service and classes that override the default behavior of the service once deployed in the Axis2 container. It is the only package that depends on the Axis2 automatically generated classes, contained in the three packages `types`, `parametersweeping` and `parametersweeping.types`, that are needed to manage the marshalling/unmarshalling of SOAP messages. The parameter sweeping Web Service core implementation is contained in the `parametersweeping.customskel` and `parametersweeping.internal` packages.

The `parametersweeping.hotspots` package defines the interfaces and abstract classes to be extended by the user in order to implement the business logic of the application according to the Parameter Sweeping pattern.

The `schedulerwsimpl.gridscheduler` package contains classes that offer interaction with the ProActive Scheduler.

The `repositoryimpl.internalrep.api` package contains classes that offer access to the archives repository.

The business logic for the operations exposed by the web service is treated at the level of `CustomParameterSweepingServiceSkeletonDelegate` class, in the `parametersweeping.customskel` package. The service needs to create a Job (ProActive Scheduling) and submit it to The ProActive Scheduler. We recall that a ProActive Job is composed of a bag of tasks. Each task can be executed independently on a remote resource.

A parallel service delegates the creation of the Job to be submitted to the Scheduler to a JobFactory. For the Parameter Sweeping Web Service the factory used is the `ParSweepingjobFactory`. For the creation of the job, the JobFactory needs: the archive containing the business code for the parameter sweeping pattern, the name of the classes that implement the hotspots, the input data, configuration data and the context info. All these information have been provided as argument, by the end client, when calling the web service.

Once the job is created by the factory, it will be sent to the ProActive Scheduler for execution. An event handler will also be sent to the Scheduler in order to receive an event when the job is finished.

If the operation has been triggered synchronously, via the `ParameterSweepingPortType` (see [Section 2.3.4, "The ParameterSweeping Functionalities"](#)) the handler will wake the calling thread in order to send the response to the end client, when a "job finished" event is received.

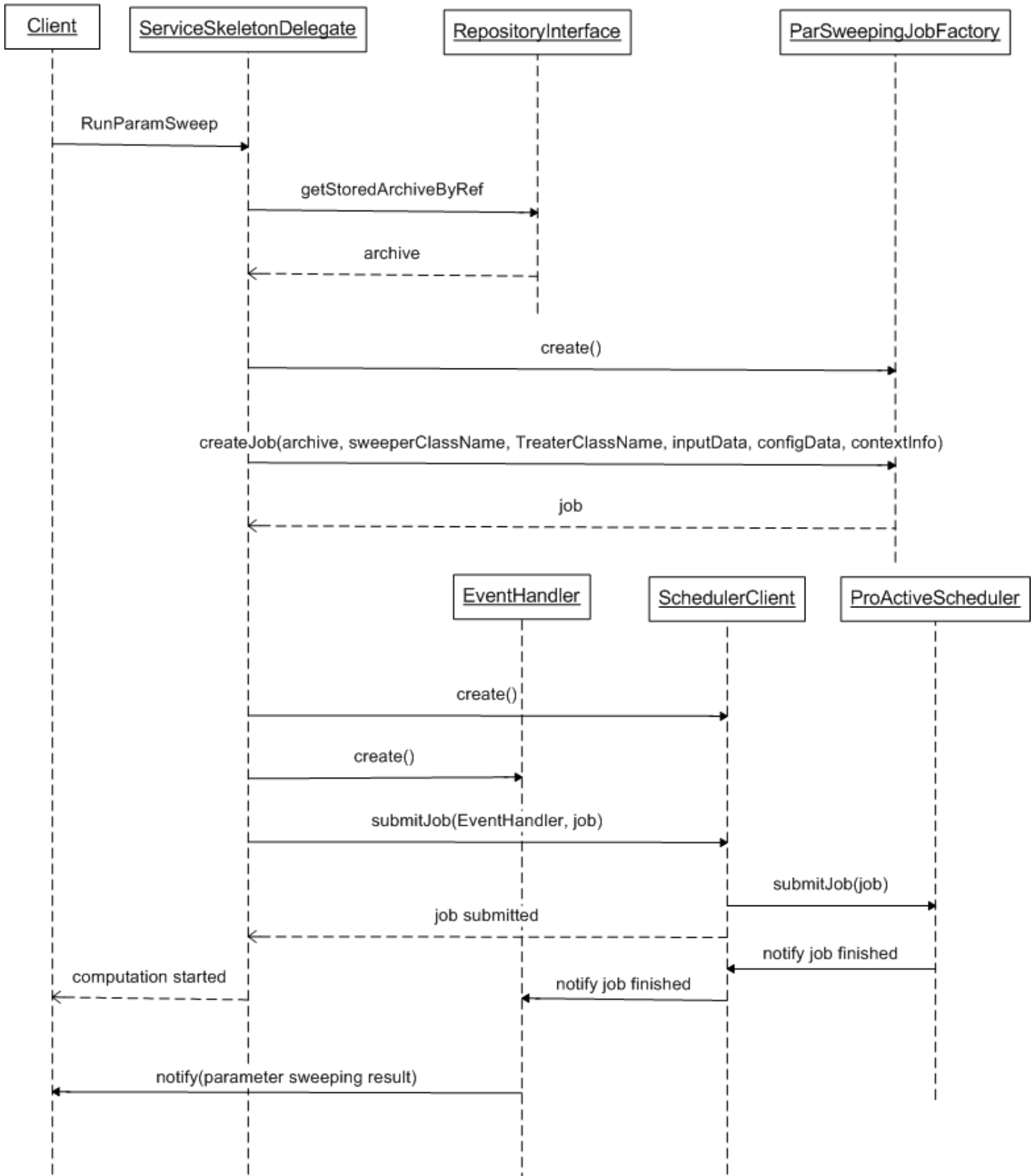
For the asynchronous call, the Handler is responsible to contact the end client and send the response when a "job finished" event occurs.

Below, a sequence diagram illustrating the asynchronous call to the Parameter Sweeping Web Service.



### Note

Not all the entities involved into the process are shown in the diagram.



**Figure 3.8. Parameter Sweeping invocation scenario - asynchronous**

The `ParSweepingJobFactory` is responsible for calling the concrete implementation of the Sweeper hotspot. The factory will use the output of the Sweeper in order to create input arguments for Tasks for the Job. The business code for each task is given by the concrete implementation of the second hotspot, the `DataSliceTreater`.

# Chapter 4. Installation

## 4.1. Introduction

ProActive Parallel Services have been developed using [Apache Axis2](#)<sup>1</sup> Web Services container. The actual distribution of Parallel Services is based on [Axis2 1.5.3](#)<sup>2</sup> version and they use the release 3.0.0 of [ProActive Scheduler / Resource Manager](#)<sup>3</sup>.

## 4.2. Installation of the infrastructure

To run the services, the following components are needed:

- an Axis2 container running the Parallel Services
- a ProActive Scheduler / ProActive Resource Manager installation
- ProActive nodes where to schedule jobs produced by ParallelServices

To install and configure the environment you need to perform the following operations:

- **Install Tomcat and Axis2** - Consider to use a Tomcat Servlet container running the Axis2 Web service container. Once [Tomcat](#)<sup>4</sup> installed, install [Axis2 in Tomcat](#)<sup>5</sup>. The Parallel Services need the support for WS-Security and WS-SecurityPolicy. Rampart is the Axis2 module providing support for WS-Security, WS-SecurityPolicy, WS-SecureConversation, and WS-Trust, so you must install Rampart in Axis2. Actually our reference version for Rampart is the [Rampart 1.5](#)<sup>6</sup> version. Rampart comes with several .jar files (in the distribution's lib directory), along with a pair of .mar module files, rampart.mar and rahas.mar (in the dist directory). To install Rampart in Axis2, just directly copy the .jar files from the Rampart lib directory into your Axis2 WEB-INF/lib directory, and the .mar files from the Rampart dist directory into your Axis2 WEB-INF/modules directory.
- **Configure Tomcat to run on HTTPS** - The services are made secured following the WS-Security standard, and they implement the simplest WS-SecurityPolicy: UsernameToken on SSL. With an Axis2 container running in a Tomcat instance, the Tomcat instance has to be configured to run on https. (For more information, see the following [howto](#)<sup>7</sup>).
- **Configure security policy for Tomcat** - Since the services' implementation is based on ProActive, they have to be run using a security manager configured with a security policy compatible with the one required by ProActive. (As a sample policy, replace the file "\$TOMCAT\_HOME/conf/catalina.policy" into your Tomcat installation with the sample catalina.policy file provided in the Parallel Services distribution).
- **Install the ProActive Scheduler / Resource Manager** - Follow the [ProActive Scheduler / Resource Manager](#)<sup>8</sup> guide to choose your configuration for the Grid infrastructure. Anyway, for a first sample test, you could install the ProActive Scheduler on the same machine of your Tomcat installation, or even on a remote one, and use the basic default configuration of the ProActive Scheduler / Resource Manager.

## 4.3. Web Services Configuration and Deployment

At this point you can configure and deploy the Parallel Services.

Parallel Services are distributed as a service archive, having the Axis2 service archive format. The `ParallelServices.aar` archive contains all the ProActive Parallel Services to be deployed as an Axis2 servicegroup.

Since the configuration for the services is contained inside the service archive, before deploying the archive you need to modify some of the following configuration files contained in the archive (uncompress the archive to edit them):

<sup>1</sup> <http://ws.apache.org/axis2/index.html>

<sup>2</sup> <http://axis.apache.org/axis2/java/core/docs/toc.html>

<sup>3</sup> [http://proactive.inria.fr/index.php?page=release\\_notes](http://proactive.inria.fr/index.php?page=release_notes)

<sup>4</sup> <http://tomcat.apache.org/download-55.cgi#5.5.28>

<sup>5</sup> [http://axis.apache.org/axis2/java/core/docs/installationguide.html#servlet\\_container](http://axis.apache.org/axis2/java/core/docs/installationguide.html#servlet_container)

<sup>6</sup> <http://ws.apache.org/rampart/download/1.5/download.cgi>

<sup>7</sup> <http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html>

<sup>8</sup> [http://proactive.inria.fr/release-doc/Scheduler/multiple\\_html/Overview.html#scheduler\\_installation](http://proactive.inria.fr/release-doc/Scheduler/multiple_html/Overview.html#scheduler_installation)

- **log4j.properties**: to configure the services' logging
- **rep.properties**: configuration properties for PSRepository service. It contains essentially configuration information about the Data Base and the physical location of the archive files.

Here is a sample configuration file, with details about each property:

```
# [changel]
# the directory where the PSRepository service
# will store all the archive files uploaded on the server side
#
fileRepositoryPath=/path/to/parallelservices/filearchives-dir/

# [changel]
# Since the Derby DB is actually used,
# this directory will contain the Derby DB database directory
#
databasePath=/path/to/parallelservices/database-dir/

# [dontChangel]
# the dbms driver. Actually only Derby has been tested
#
driver=org.apache.derby.jdbc.EmbeddedDriver
protocol=jdbc:derby:

# [dontChangel]
# The name of the Parallel Services Data Base
# This DB is used to store all the information about the
# archives stored on the server side via the PSRepository service
#
databaseName=PSREPOSITORY_DB
user=ps_user
password=ps_password
```

- **schedconf.properties**: configuration properties needed to interact with the ProActive Scheduler and configuration properties specific to the SchedulerInterface service.

Here is a sample configuration file, with details about each property:

```
                # [changel]
# The URL of the ProActive Scheduler
#
schedulerUrl=rmi://localhost:7777/

# [changel]
# Credentials of a ProActive Scheduler's user, used by the
# services' logic to monitor the Scheduler job execution
#
usr=demo
psw=demo
```

```

# [changel]
# The directory used by the SchedulerInterface as a
# work directory where to extract the job archive of
# a job submitted via the SchedulerInterface service.
# (It should be reachable via NFS from all the
# computational nodes, if is adopted a policy where
# job descriptions admit scripts and resources
# accessed at runtime)
#
jobInstancesDir=/path/to/parallelservices/PS_WORK/

# [optional]
# Path to the ProActiveConfiguration.xml file used to
# configure ProActive middleware in the services logic.
# If not set, the default ProActive configuration is used
#
#proactiveConfigurationFile=/path/to/parallelservices/srcproject/proactive/ProActiveConfiguration.xml
proactiveConfigurationFile=

# [dontChangel]
# synchronization delay in case of asynchronous interaction
#
callbackDelay=3000

# enabling ws-security
wsSecurityEnabled=true

```

- **paramsweeping.properties**: configuration properties specific to the Parameter Sweeping service.

Here is a sample configuration file, with details about each property.

```

# [changel]
# ProActive Data Space enabled
# If true, files created by the business logic
# during the Sweeping phase will be automatically
# transferred to the computational nodes
# If NFS is available between the application server
# and computational nodes, that's not necessary
# [ possible values : true/false ]
#
enableDataSpaces=false

# [optional]
# Path on the file system accessible from the ParameterSweeping service.
# In this folder business logic will create files, if needed.
# If data space is enabled, files in this folder can be accessed from remote nodes.
# Leave it empty in order to use the application server's temp folder

```

```

# Note: if dataspace are not enabled, this folder should be accessible,
# via nfs, from the computing nodes
#localUserWorkingFolder=/path/to/paramsweeping/dataspace/dir
localUserWorkingFolder=

# [optional]
# The maximum number of executions of each
# Data Slice Treater task, in case of task failure
# If empty, the default value is '3'.
#
taskMaxNumberOfExecutions=3

```

Once you have modified these configuration files, re-create the archive with the modified configuration files (just compress everything one more time), and **deploy** the services group (just copy the archive `ParallelServices.aar` under the Axis2 services repository, the `WEB-INF/services` directory).

## 4.4. Run the Parallel Services

Once you have installed and configured everything as said above, you are ready to run the services.

- **start the ProActive Scheduler** - (In a sample run, if you start the `ProActiveScheduler` without any option, by default it will look for an instance of the `ProActive Resource Manager` running on the same host, otherwise it will launch one). Using the `ProActive Resource Manager` user interface, you can see, manage and add `ProActive` nodes that represent the Grid infrastructure resources to be used to schedule jobs on.
- **deploy the services** - if you have not yet done it (see the section [Section 4.3, “Web Services Configuration and Deployment”](#)). *NB: if the services have already been deployed before, deploying a new version of the services needs the restart of the Tomcat server, because of some initialization operations done at deployment time.*
- **run Tomcat** - using a security manager (for example, on Unix, run `$TOMCAT_HOME/bin/startup.sh -security`). Now Axis2 should be running and all the Parallel Services should be ready. Check that they are actually ready, accessing for example to: `https://localhost:8443/axis2/services/listServices` or, to see the wsdl of a specific service, check for example `https://localhost:8443/axis2/services/PSRepository?wsdl` (if the server is running on localhost).

## 4.5. Web Service building from sources

Once the project is checked out, its contents is organized as described in the following extract of its `readme.txt` file.

### PROJECT CONTENTS

-----

The project is organized as follows:

```

./src/parallelservices/src/repository : src for the PSRepository service
./src/parallelservices/src/scheduler  : src for the SchedulerInterface service
./src/parallelservices/src/paramsweeping : src for the ParameterSweeping service
./src/parallelservices/src/common     : src commons to all the other services (or some of them)
./src/parallelservices/src/utills     : src for utilities classes
./src/parallelservices/tests         : src for tests
./src/parallelservices/client        : src for client stub and sample client code

./compile :

```

- lib, formatlib, eclipse\_formatter\_config.properties and eclipse\_formatter\_config.xml:  
libs and ant artifacts specific for ant target execution
  - build-res: contains everything needed to compile and build the deployment version of the Parallel Services, such as:
    - configprop: contains all the configuration files to configure the services
    - pa\_stub\_generation: contains a build file used to generate the ProActive stubs needed by the application
    - servicesxml: contains the services.xml file to build the deployment archive for the Parallel Services groupThis directory will contain also all the resources automatically generated by the building process:
    - dist-tmp: this directory is created to contain the distribution archives, that will be generated by the building process.
    - classes: all the class files, except test classes
    - test-classes: all the test class files
    - output: all the source files automatically generated (by the build process using wsdl2java tool)
    - junitReports: all the junit report files generated by junit tests when launched via the building process
  - build.xml : the main ant build file for building process.  
It imports all the other ant files. All the ant targets MUST be started running ant from this build file.
  - build.properties : it contains the properties that are specific to the user environment and that should be changed in order to perform some of the building steps.
  - build : executable script to use to run ant on Linux platforms
  - build.bat : executable script to use to run ant on Windows platforms
- ./dist : the directory that will contain all the distribution files
- ./doc : contains all the resources needed to generate the project's documentation.  
**IMPORTANT:** this directory will contain all the generated documentation, in a 'dist' directory automatically created by the building process.
- ./jobArs : contains some sample job archives.  
They have been built starting from the samples of the ProActive Scheduler distribution, adapting, where necessary, the job descriptor and the needed resources in order to build correctly the corresponding job archive. (These archives can be used as testing/sample archives, since they can be stored in the repository, via PSRepository service, and can be submitted to the ProActive Scheduler via the SchedulerInterface service, using their reference in the repository)
- ./lib/junit : contains the junit jar files
- ./lib/4WSRep : contains all the jar files needed to manage and access the repository
- ./lib/currentSchedDistLib : contains all the jar files of the Scheduler distribution currently used
- ./lib/axis2\_rampart : contains all the jar files of the axis2/rampart installation actually used
- Actually these set of jar files is composed of :
- all the Axis2 1.5.3 jar files
  - all the Rampart 1.5 jar files (Rampart is the module providing WS-Security support for Axis2)
- ./proactive: contains a sample ProActiveConfiguration.xml file
- ./psArs : contains sample parameter sweeping archives.  
(These archives can be used as testing/sample parameter sweeping archives, since they can be stored in the repository, via PSRepository service, and can be submitted to the ProActive Scheduler, via the ParameterSweeping service)

```
./test-res : contains resources needed for test execution  
./wsdl : contains all the WSDL and XSD files for all the parallel services.
```

Check the readme.txt file in order to have all the information related to the source project.

Once the check-out is complete, in the whole project, several source files are missing. The development of each service follows the WSDL-first approach. Starting from the service contract, such as the WSDL definition of the service, (contained in the `./wsdl/parallel_services` dir) the skeleton sources are automatically generated by the `wsdl2java` tool provided by Axis2 (that's why these source files are not under version control, while the WSDL files are). This code is generated launching the ant target `cleanAndGenerate.all` (Each time the operation is performed, the old generated sources are replaced by the new ones) Once done that, the source code is complete and ready to build the services to deploy. **BE CAREFUL**: all the ant targets have to be run starting from the main build file: `./compile/build.xml` and including in the ant classpath the jar files contained in the `./compile/lib` directory.

In the source project, you can set the configuration files we have spoken about in [Section 4.3, “Web Services Configuration and Deployment”](#) editing them in the directory `./compile/build-res/configprop`. Once done, you can automatically build the services archive just running the `aar` ant target. It will produce the archive `./compile/build-res/dist/ParallelServices.aar` that you could deploy under Axis2 as said above. Otherwise, if your Axis2 container is already installed and it is on the local machine (or on a machine where its folder is reachable via file system) you can directly build and deploy the parallel services group under your Axis2 installation. To do that:

- modify in the file `./compile/build.properties` the property `TOMCAT_HOME` related to the "Axis2 installation"
- run the `deployOnServer` ant target. **NB: if the services have already been deployed, this operation performs their undeployment before deploying the new version.**

At this point you could switch on all the configured servers and test the Parallel Services.