Deliverable Title	**D4.2.1 – Experiment and Workflow Scheduler prototype**
Deliverable Lead:	ActiveEon
Related Work package:	WP4
Author(s):	Cedric Dalmasso, Franca Perrina
Dissemination level:	Public
Due submission date:	30/04/2011
Actual submission:	30/05/2011
Project Number	258142
Instrument:	IP
Start date of Project:	01/06/2010
Duration:	30 months
Project coordinator:	THALES

Abstract	This document gives an overview of the prototype delivered in the task D4.2.1. It presents the main functions provided by the delivered solution and it gives installation instructions and details about the prototype running in the TEFIS infrastructure operated by INRIA.


Project funded by the European Commission under the 7th European Framework Programme for RTD - ICT theme of the Cooperation Programme.

License

Versioning and Contribution History

Version	Date	Modification reason	Modified by
0.1	20/04/2011	First Draft	Cedric Dalmasso, Franca Perrina
1.0	30/04/2011	Final version	Cedric Dalmasso, Franca Perrina

TABLE OF CONTENT

1. Executive Summary 6

2. Introduction 6

3. Experiment and Workflow Scheduler overview 8

3.1. Novelties and design decisions 8

3.2. The ProActive Scheduling tool 10

3.2.1. The ProActive job 12

3.2.2. The Task definition 15

3.2.3. The ProActive Scheduling interfaces 17

3.3. The proposed solution 18

3.3.1. The TCI-EXEC API 18

3.3.2. The Task Templates 20

3.3.3. The experiment definition and execution 23

3.4. TEFIS specific developments 24

4. Prototype installation 26

4.1. Scheduler server 26

4.2. Scheduler REST API 27

5. Conclusion 28

References 30

TERMINOLOGY

Scheduler	The TEFIS Core Service named “Experiment and Workflow Scheduler”
Test plan	The definition of the experiment in terms of tasks, or process steps, and resources involved and associated information
Test run	The definition of a specific run of the experiment. It represents a specific instantiation of the experiment test plan

Task	Atomic step of an experiment test plan or test run
Process step	Synonym of task
TCI	TEFIS Connectors Interface
TCI-EXEC API	TEFIS Connectors Interface - Application Programming Interface for EXECution

1. Executive Summary

This document arguments the reasoning and the choices behind the implementation of the Experimental and Workflow Scheduler component that is part of the TEFIS platform architecture as stated in the deliverable *D2.1.1 – Global architecture and overall design*.

Firstly, in section 3, we identifies the features and requirements the Experimental and Workflow Scheduler must have and we argument the choice of the existing open source product, ProActive Parallel Suite and in particular of the ProActive Scheduling tool contained in that suite, to match those features and requirements.

In section 3.2, we present the details of the ProActive Parallel Suite: its overall architecture, the interfaces it exposes (Java API, JMX), what is a ProActive job and a ProActive task and how to define them.

Then, in section 3.3, we move on the description of the solution about the Experimental and Workflow Scheduler proposed in the TEFIS platform. We describe and how that component enacts the experiment the user has defined. We include the description of the interactions with the testbed connectors and how the Experimental and Workflow Scheduler leverages the TCI-EXEC API to execute a process step of the experiment the user want to run. In the same section we introduce the concept of *Task Template* too as a mean to deal with the different kinds of activities the user can perform on the same testbed. The Task Template allow to use the same ProActive task definition but with different parameters to execute different activities on the same testbed. Finally, we talk about the extensions the ProActive Scheduling tool has required to fully match the requirements and features identified for the TEFIS Experimental and Workflow Scheduler.

In the last section of the document, 4, we describe how to install the Experimental and Workflow Scheduler, how to start and stop it and some commands that allow job management (e.g., look at the state of the ProActive job, get the result of a ProActive task etc…). In that section we also present the ProActive Scheduling tool REST API.

2. Introduction

This document serves as a supporting document for the Experiment and Workflow Scheduler prototype delivery. It aims to give a general idea of the main functionalities provided by the delivered solution, as well as information about the prototype currently installed within the TEFIS platform.

It is structured as follows:

- the first section gives a functional overview of the system, starting from an introduction about the ProActive Scheduling tool used as a basis for the solution and then presenting the extensions and the modules added to build the current solution;
- the second section covers aspects mainly related to the installation and configuration of the system on the TEFIS infrastructure.

The delivered prototype, at the time of writing, is installed in the TEFIS infrastructure hosting the first release of the TEFIS platform.

3. Experiment and Workflow Scheduler overview

The Experiment and Workflow Scheduler is the TEFIS Core Service in charge of the orchestration of the experiments execution over the available testbeds (cf. the deliverable D2.1.1 [1] for a global view of the functional architecture of the TEFIS platform).

The Core Services components are in charge of all the main interactions with testbeds, through the TEFIS connectors. The Resource Manager is the Core Service ([2]) in charge of the management of the heterogeneous resources to be provisioned for an experiment. The Experiment and Workflow Scheduler (called 'Scheduler' or 'TEFIS Scheduler' for brevity, in the rest of this document) is in charge of the enactment of the experiment flows designed by the experimenters with the appropriate tools provided by the TEFIS Portal.

The solution proposed for the Scheduler implementation is based on the ProActive Scheduling tool [6]. The following section, 3.1, explain the novelties and the design decisions behind the TEFIS Scheduling, section 3.2 proposes a general overview of the tool, while the section 3.4 presents the main extensions realised to obtain the TEFIS Experiment and Workflow Scheduler solution.

3.1. Novelties and design decisions

The TEFIS platform provides to the experimenter the possibility to define his experiment through a zero code experiment design process: the experimenter defines only the graphical model and some configuration information for his experiment and doesn't write any code using a programming language. The experiment defined by the user is a high level workflow. It only represents and specifies the sequence of activities and resources on which those activities have to be performed. Activities and resources are testbed specific. In other words, the information required and parameters to give values at to define activities and resources vary from testbed to testbed. Anyway it's not up to the user to discover which information and which parameters each testbed requires. It's the TEFIS platform that supports the user during the definition of the testbed specific information and parameters. In particular, it's the TEFIS platform that requests to the user to set up different set of parameters depending on which is the testbed that provides the needed resource. That means for each testbed (and so resource provided) TEFIS require to the user the definition of a different set of information and/or parameters.

The user has not to take care of the location of testbeds involved in his experiment and about how to access them (e.g., which protocol must be used). Hence, the user has not to specify the details related to the access of the testbeds when he defines the high level workflow. Those information are only known at testbed connector level because only the connectors can interact with the testbeds.

The high level workflow defined by the user is not executable directly, but it is needed that TEFIS, through some modifications, turns it into a fully executable workflow. To do that, TEFIS has to preserve the structure of the high level workflow, create a low level workflow in which some missing details (e.g., about how to access the connectors of the testbeds involved in the experiment) are added and translate the resulting workflow into an executable (i.e., create a representation of the workflow in one programming language and then compile that program to produce the executable of that workflow).

Hence, the features the component, that implements the Experiment and Workflow Scheduler in TEFIS, must provide are the following:

- Be able to create a low level workflow whose structure reflect, exactly, the one of the high level workflow defined by the experimenter;
- The translation from the high level workflow into the executable must be easy. That means the work required by the TEFIS developers to do that must not be too hard;
- Add all the necessary and missing information (e.g., information about how to access the testbed connectors) to the high level workflow to turn it into an executable;
- Enact the executable low level workflow;
- Low level workflow tasks must include the logic to interact with the testbed connectors;
- Define as many low level tasks as testbed connectors. In fact, in TEFIS there are different connectors, one for each testbed plugged into the TEFIS platform. Testbed connectors implement the same interface, the Testbed Connector Interface (TCI), but their implementations differ because each of them has to reflect the interaction specifities of a particular testbed. This means each testbed has its own connector and there is a different low level task for each connector;
- Define different "task templates", one for each testbed. The task template can be seen as the combination of the low level task and its own set of configuration parameters (testbed connector to contact, resources involved in the execution of the activity that task represents etc…). In other words there is one task for each testbed connector and each task can be configured differently depending on the activity it has to perform on the testbed it's related to. That means the task of the low level workflow should be parametrizable;
- Workflow tasks should be linked among them, so that beyond the flow of execution there should be a flow of data too. That means the transfer of data from a task towards the one that follows should be done automatically without the user's intervention;
- The workflow has not to be enacted on a single machine, but the tasks must be distributed to different machines to balance their execution load. This point is important because it allows TEFIS to be scalable: if the number of workflows (and so experiments) increases then some more machines can be added to avoid overloading the existing ones;
- Different platforms (UNIX/LINUX, Mac, Windows) must be supported. That means that when the Experiment and Workflow Scheduler has to be scaled the administrator of the TEFIS platform is not constrained to add only a particular platform;
- Allow the monitoring of the execution of the workflow (e.g., what is the overall state of the running workflow, which step is currently executed etc…).

The first constraint we take into account for choosing the eventually existing component that provides the features listed above is the following concept claimed in the TEFIS Dow (Description of Work): "All

the TEFIS software and components will be released with an Open Source license. This licence could be GPL, LGPL, BDS or EUPL. The choice will depend of the compatibility between all the components. By default, the AGPL will not be used.". That means among the set of known and available tools that match the description of the component Experiment and Workflow Scheduler required in the TEFIS platform we filtered out commercial products and the products that are not distributed under one of the admitted licenses.

There is more than one open source product that we can choose to match the requirements expressed above: DIET[1] (Distributed Interactive Engineering Toolbox), Job2Do[2], Open Source Job Scheduler[3], Open PBS (Open Portable Batch System) that is the open source version of PBS (Portable Batch System)[4], ProActive Parallel Suite[5], Quartz[6] and SLURM[7] (Simple Linux Utility for Resource Management). Among them we choose the ProActive Parallel Suite as the component that will implement the Experimental and Workflow Scheduler. It is made up of Java grid middleware for parallel, distributed, and multi-threaded computing, a batch job scheduler and a resource manager. ProActive Parallel Suite matches most of the identified requirements. The only feature it does not provide is the concept of "Task Template". But as it will be clear from sections 3.2 and 3.4 that feature can be easily implemented. Hence, the only work required to adopt ProActive in the TEFIS platform is the definition of the algorithm to translate the high level workflow in the low level one. The low level workflow, represented using XML, can be directly managed and executed by the ProActive Scheduling tool.

The sections that follow go deeply into the description of the ProActive Parallel Suite and of the ProActive Scheduling tool in particular and it allow to better understand why ProActive is a suitable and valid solution to implement the Experiment and Workflow Scheduler component of the TEFIS platform.

3.2. The ProActive Scheduling tool

The ProActive Scheduling tool is a batch scheduler for execution of jobs on a shared set of computing resources. The ProActive Scheduling tool enables users to submit jobs, containing one or several tasks, and then to execute these tasks on available computing resources. It allows several users to share a same pool of computing resources and also to manage issues related to distributed environments, such as failing resources. The ProActive Scheduling tool is connected to the ProActive Resourcing tool that provides therefore the computing resource abstraction [2].

The picture below, Figure 1, shows the overall architecture of the ProActive Scheduling tool.

---

[1] http://graal.ens-lyon.fr/DIET/
[2] http://www.sypsoft.com/job2do.html
[3] http://www.sos-berlin.com/modules/cjaycontent/index.php?id=osource_scheduler_introduction_en.htm
[4] http://www.pbsworks.com/?AspxAutoDetectCookieSupport=1
[5] http://proactive.inria.fr/
[6] http://www.quartz-scheduler.org/
[7] https://computing.llnl.gov/linux/slurm/

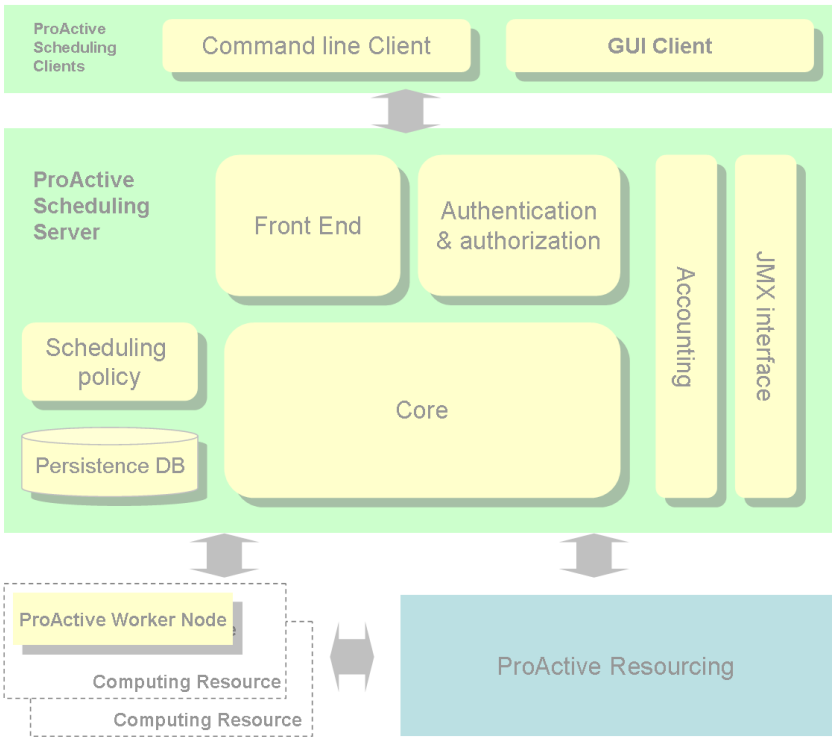

Figure 1 ProActive Scheduling overall architecture

The picture shows the ProActive Scheduling main functional entities. The ProActive Resourcing tool is also reported in the pictures to show that the Scheduling server leverages the ProActive Resourcing tool to gain access to available Computing Resources. A Computing Resource is a physical resource hosting a ProActive entity, the ProActive Worker Node in the picture, which represents the entry point on the physical resource, to let the Scheduler server launch a task execution on the resource.

The architecture of the ProActive Scheduling Server is built around three main components:

- Authentication & authorization: it is in charge of authenticating and authorizing the users to access (or not) the Scheduling services. The authentication security system can interact with files or LDAP systems;
- Front End: it allows users to submit jobs, get scheduling state, retrieves job result, etc.
- Core: is the main entity of the ProActive Scheduling server. It is in charge of scheduling jobs according with the Scheduling policy, managing delivery of scheduling events to subscribed users, managing persistence of the scheduling process.

The server includes other relevant components, such as:

- Scheduling policy: it is the algorithm used by the Scheduler to order jobs and tasks execution. It is a pluggable entity so that the scheduling behaviour can be adapted to a specific instantiation of the system.
- Accounting component: it is in charge of tracking how many jobs and tasks have been initiated by the users, the overall amount of time spent in job execution by the users, etc.

- JMX Interface: it is a remote management and monitoring interface, based on the JMX standard, providing information about the status of the system or about the total number of jobs and tasks in a given status, but it gives also access to accounting metrics and to various management operations. This interface is accessed by a ProActive Scheduling Client but also by a standard JMX client.

On the client side a Command Line Client and a Graphical User Interface are provided to the users to let them remotely interact with the server.

3.2.1. The ProActive job

In the ProActive Scheduling system a Job is the set of tasks to be submitted to the scheduler for execution. It is composed of a bag of tasks, which can be executed either in parallel or according to a dependency tree and common control flow structures (such as *for*, *while*, *if-then-else*). The picture below, Figure 2, reports a graphical representation of a sample job.

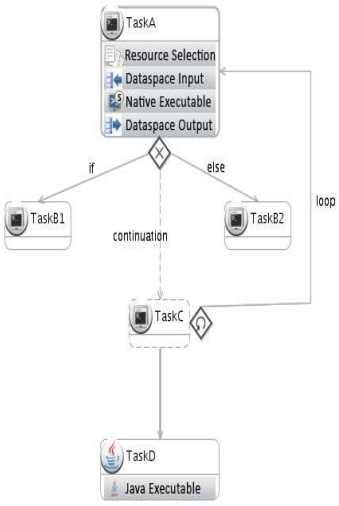

Figure 2 A ProActive Job

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 for example an MPI application (execution starts with a given predefined number of resources). The job is represented in an XML format, according to a well defined schema [8]. Below a sample fragment is reported.

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="…schedulerjob.xsd" cancelJobOnError="true" name="example" priority="low">
    <variables>
       <variable name="val1" value="my value" />
       <variable name="convert_ex" value="/usr/bin/convert" />
    </variables>
    <description>the job description..</description>
    <jobClasspath>
       <pathElement path="/path/to/my/classes/" />
       <pathElement path="/path/to/my/jarfile.jar" />
    </jobClasspath>
    <genericInformation>
       <info name="var1" value="${val1}" />
       <info name="var2" value="val2" />
    </genericInformation>
    <inputSpace url="ftp://myserver.org/myjob/input" />
    <outputSpace url="ftp://myserver.org/myjob/output" />
    <taskFlow>
       <task name="TaskA">
              <inputFiles>
                     <files includes="im.sh" accessMode="transferFromInputSpace" />
                     <files includes="00?.jpg" accessMode="transferFromInputSpace" />
              </inputFiles>
              <selection>
                     <script>
                            <file url="http://proactive.inria.fr/userfile../checkExecutable.js">
                                   <arguments>
                                          <argument value="${convert_ex}" />
                                   </arguments>
                            </file>
                     </script>
              </selection>
              <nativeExecutable>
                     <staticCommand value="/bin/sh" workingDir="$LOCALSPACE">
                            <arguments>
                                   <argument value="im.sh" />
                                   <argument value="output_%02d.jpg" />
                            </arguments>
```

```
                        </staticCommand>
                </nativeExecutable>
                <controlFlow block="start">
                        <if target="TaskB1" continuation="TaskC" else="TaskB2">
                                [… ]
                </controlFlow>
                <outputFiles>
                        <files includes="output_*.jpg" accessMode="transferToOutputSpace" />
                </outputFiles>
        </task>
        <task name="TaskD">
                <depends>
                        <task ref="TaskC" />
                </depends>
                <javaExecutable class="org.package.MyApplication">
                        <parameters>
                                <parameter name="inputParam1" value="6" />
                                <parameter name="inputParam2" value="7" />
                        </parameters>
                </javaExecutable>
        </task>
    </taskFlow>
</job>
```

Several parameters can be set for a job. Few of them are reported below (cf. [6] for an exhaustive list):

- name: name of the job;
- priority: priority level (such as 'low', or 'normal', ‘high’, etc.);
- cancelJobOnError: it defines whether the job must continue when a user exception or error occurs during the job process;
- description: human readable description of the job;
- variables: variables which can be reused throughout the descriptor file;
- genericInformation: defines some information inside the job. This information can be read by the scheduling policy that can then modify the scheduling behaviour;
- jobClasspath: it is equivalent to the Java classpath and all classes in these paths can be loaded by Java tasks contained in the job;

- inputSpace, outputSpace: URLs that define the INPUT/OUTPUT spaces of the job. The INPUT/OUTPUT space URL represents an abstract (or real) link to a real set of data. It is used for data transfer and offers a way to get, from the INPUT space, files to be computed on the task execution side and to put produced files from the task execution side to the OUTPUT space (for details).

The sample job fragment reported above shows two tasks.

The task named "TaskA" presents four sections, identified by the associated XML tags. The 'inputFiles' section defines the files that are expected to exist in the provided 'InputSpace' and that the system should automatically transfer ('accessMode="transferFromInputSpace"') from there to a scratch space on the computing resource in charge of the task execution. The 'selection' section defines a way to select a computing resource able to execute the task (in the example, a script checks if the expected executable exists on the computing resource). The 'nativeExecutable' section defines the actual native task to execute (in the example the user script 'im.sh' retrieved from the input space is executed on the resource using the scratch space – '$LOCALSPACE' - as working directory). The 'controlFlow' section defines the control flow structure that follows the task (in the example it is an 'if-then-else' control flow structure, but it is not completely reported here for brevity). The 'outputFiles' section defines the files that are expected to be produced in the scratch space by the task and that the system should automatically transfer ('accessMode="transferToOutputSpace"') to the output space.

The task named "TaskD" presents two sections. The section 'depends' defines the dependencies on other tasks, meaning that this task has to be scheduled after the tasks it depends on (in the example, it depends on 'TaskC', not reported in the fragment). The 'javaExecutable' section defines the Java task that has to be executed. The user class representing the task implements a well known interface and this class, together with the classes and libraries it depends on, have to be included in the 'jobClasspath' section specified for the job, so that the system will be able to load the classes needed for the task execution. The object instantiated by the system will be initialized with the 'parameters' defined by the user in this section.

This sample job gives only a partial idea of what a job can look like. More details are provided in the user manual of the ProActive Scheduling system [6].

3.2.2. The Task definition

In a ProActive job, a Task is the smallest schedulable entity. It is included in a Job and will be executed in accordance with the scheduling policy on the available computing resources. There are two types of tasks:

- JAVA task: its execution is defined by a Java class extending the org.ow2.proactive.scheduler.common.task.executable.JavaExecutable class;

- NATIVE task: its execution can be any user program, a compiled C/C++ application, a shell or batch script (it 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).

In this presentation we focus mainly on JAVA tasks, since we will use them in the proposed solution (a complete description of the two types of tasks is given in [6]). A user task has to extend a well known `Executable` class, but the `JavaExecutable` abstract class is already provided, so the user can start extending the `JavaExectutable` class. Below is reported an extract of each of them.

```
package org.ow2.proactive.scheduler.common.task.executable;
[...]
/**
 * Executable is the superclass of every personal executable task that will be scheduled
 * [...]
 */
@PublicAPI
public abstract class Executable {
    /**
     * The content of this method will be executed once after being scheduled.<br>
     * This may generate a result as an {@link Object}. It can be whatever you want.<br>
     * The results list order corresponds to the order in the dependence list.
     *
     * @param results the results (as a taskResult) from parent tasks.
     * @throws Throwable any exception thrown by the user's code
     * @return any serializable object from the user.
     */
    public abstract Serializable execute(TaskResult... results) throws Throwable;
[...]

}
```

Below the `JavaExectutable` class, that extends the previous one.

```
package org.ow2.proactive.scheduler.common.task.executable;
…
@PublicAPI
public abstract class JavaExecutable extends Executable {


    /**
     * Initialization default method for a java task.<br>
     * <p>
     * By default, this method does automatic assignment between the value given in the arguments
     * map and the fields contained in your executable.<br>
     * If the field type and the argument type are different and if the argument type is String
     * (i.e. for all jobs defined with XML descriptors), then an automatic mapping is tried.
     * Managed types are byte, short, int, long, boolean and the corresponding classes.<br><br>
     * For example, if you set as argument the key="var", value="12" in the XML descriptor<br>
     * just add an int (or Integer, long, Long) field named "var" in your executable.
     * The default {@link #init(Map)} method will store your arguments into the integer class
field.
     * </p>
     * To avoid this default behavior, just override this method to make your own initialization.
     *
     * @param args a map containing the different parameter names and values given by the user
task.
     */
    public void init(Map<String, Serializable> args) throws Exception {
[...]
    }
[...]
}
```

The user defined Java task has to extend the abstract class reported above. The java task represents a java process as a java class. The main method to implement is the `execute` method, that contains the main logic of the task. The `init` method is used to initialize the task with specific parameter values. The default behaviour of this method assigns automatically the values passed in the given map `args` to the fields defined in the user class and which have the same names as the map's keys. The method can be overridden to change this behaviour.

3.2.3. The ProActive Scheduling interfaces

The ProActive Scheduling system implements several functions that are exposed to external clients. As seen above, several management and monitoring functions are exposed by the JMX Interface, but the main functions are exposed as a Java API and they are:

- job submission;
- job status monitoring;
- job and task control (pause/resume/kill ..);
- priority management;
- result/output retrieval;
- scheduling system administration (pause/resume/stop..).

External clients can access the ProActive Scheduling functions, according to their privileges, from remote locations via this Java API. Final users can use the clients already provided with the distribution of the application. The picture above, Figure 1, shows two different clients: a command line client and a Graphical User Interface client (the latter is a standalone application built on top of Eclipse Rich Client Platform).

3.3. The proposed solution

The TEFIS Experiment and Workflow Scheduler solution is based on the ProActive Scheduling tool.

In TEFIS the main objective of the Scheduler is to enable the execution of the experiments on the testbeds connected to the TEFIS platform. A TEFIS experiment test plan is composed of several steps, each one involving resources of a specific testbed. When the user want to execute the experiment, they create a test run starting from the defined test plan, they customize the test run with specific parameters, if needed, and then they start the execution.

In this scenario, the term *resource* is used to identify an instance of a *resource specification*, as they are defined in the TEFIS Resource Directory [2]. In practice, a resource is whatever is provisioned by a testbeds, in order to allow the user execute their experiments on the testbed itself. These resources can be very different with each other, but the users who define the test plan, select the needed resources to involve in the experiment and should define the tasks that make use of these resources appropriately.

In the first release of the TEFIS platform the steps included in the plan are structured as a sequence of steps, where the step N depends on the step N-1. In future TEFIS releases more complex steps structures may be included in an experiment test plan. The current version of the ProActive Scheduling tool, as said, supports basic dependencies structures between tasks but also more complex ones.

Each testbed is connected to the TEFIS platform via a Connector compliant with the TEFIS Connector specifications [7]. The set of API defined by the specifications includes the TCI-EXEC API for experiment execution. The objective of the TEFIS Scheduler is to enable the execution of the experiments, defined as a set of steps, on the testbeds, leveraging their TCI-EXEC API. The TCI-EXEC API exposes methods to run custom *executable* scripts in the testbed. The API is detailed in [7], section 5.4, but it is reported here for convenience.

3.3.1. The TCI-EXEC API

All the TCI APIs rely on a common entity, the `Experiment` entity, a structure that refers to the experiment in the TEFIS System that is concerned with the request run at the testbed and that holds all

relevant information of the experiment. It is a sort of context object used to pass to the connector a set of information that characterizes the experiment.

The most important attribute of the `Experiment` structure is the `experimenter` attribute. `experimenter` is set to the TEFIS account name of the experimenter who is running the experiment. The importance of this attribute resides in the fact that it is used by the connector to select the right user identity for the testbed domain, looking at the existing identity bindings, to access the testbed.

A possible representation of the experiment structure in xml is presented below.

```
<experiment>
	<experiment_id>testbed_uuid_string</experiment_id>
	<experimenter>john.doe</experimenter>
	...
</experiment>
```

The main method of the TCI-EXEC API is the `execute()` function which takes as a parameter an `ExecutableEntity`, the resource where the execution will take place and the experiment structure and returns an identifier that can be used later on to retrieve the status of the execution:

```
execute(executable: ExecutableEntity, resource_id: Identifier, exp: Experiment) : uuid
```

The return value is a string that represents the identifier (which is unique for the testbed) given by the connector/testbed to the job. The `execute()` method invocation is synchronous and returns as soon as a `job_id` has been assigned to the executable object.

Function `getJobStatus()` gets the `job_id` parameter and will return a `job_status` structure:

```
getJobStatus(job_id: string) : job_status
```

The *job_status* structure is pretty simple and contains two fields: *status* and *message*. The former is a string that can assume one of the values listed in ***Table 1***, the second is a free text string that can be used to transmit human readable messages.

A simple representation in xml might be:

```
<job_status job_id="…">
	<status>error</status>
	<message>ERR10: Unable to connect to the resource</message>
</job_status>
```

Table 1: possible values for the "status" field

Job_status	Description
`unknown`	The `job_id` is not associated with any of jobs known to the connector
`submitting`	The job is at the initial submission phase
`queued`	The job has been correctly received and is on a queue of jobs waiting to be executed
`running`	The job is currently running on the testbed. The message attribute may contain information like the percentage complete
`paused`	The job has been suspended
`ended`	The job terminated successfully
`error`	The job failed. The message attribute should contain the description of the error

In a typical interaction of the TEFIS System with this API, the sequence of calls will be:

1. The job is submitted by calling the `execute()` method;
2. The `job_id` string returned by `execute()` is stored in the TEFIS System;
3. The TEFIS System periodically, or when requested by the experimenter, polls the connector calling the `getJobStatus()` method to be aware of the status of the job.

3.3.2. The Task Templates

Each testbed connected to the TEFIS platform should implement the TCI-EXEC API.

The current version of the TEFIS Connector specifications does not fix the communication protocol, but since the connectors could be deployed inside or outside the TEFIS infrastructure, therefore, a remote communication protocol would need to be adopted. However, whatever is that communication protocol, the impact on the connectors interface and implementation is very small since only the format in which information are exchanged changes, not the TCI-EXEC API itself that is actually characterised by its methods signature.

Considering the specification of the API reported in the section 3.3.1, we can see that there are few items that are specific of a testbed connector implementation. The *executable* scripts, called `ExecutableEntity` in the API seen above, are the most specific part of each testbed. An `ExecutableEntity` represents a sequence of instructions that can be automatically executed on testbed resources, so this entity wraps the actual script that will be executed by the testbed's execution engine. Whatever the script is, it has to be passed to the testbed connector in order to launch the execution of the experiment on the testbed resources. In the same way, the resource `Identifier` and the

`Experiment` entity have to be defined according to their specificities, if any, and then passed to the connected testbeds during the invocation of the execution.

Summarizing, the TCI-EXEC API is common to all the testbed connectors, but the types of the `ExecutableEntity`, of the `Experiment` entity and of the resource `Identifier` are specific of each testbed connector. Anyway, it has to be remarked that, even if they are specific of each testbed connector, these types could be well known to the TEFIS system when the connector is plugged in, so the system could ask the user to fill in the expected parameters needed to customize an experiment step.

The invocation of the TCI-EXEC API of the testbeds involved in an experiment is up to the TEFIS Scheduler. As seen in the section 3.2.1, a ProActive job is composed of a set of tasks and in this context the tasks to be included in a ProActive job are the wrappers of the interactions with the involved testbeds. It means that each task is the caller of the Connector TCI-EXEC API exposed by each testbed and its role is to request the execution of the user script on the related testbed connector and check the status of the execution on the testbed using the provided API. The definition of the script to be run on the testbed at each experiment step is up to the experimenters, while the corresponding ProActive task that interacts with the testbed connector via the TCI-EXEC API is hidden to the final users, since the TCI-EXEC API is a well known API, but still internal to the TEFIS system.

As previously said, the TCI-EXEC API is common to all the testbed connectors, so the ProActive tasks needed to call the connectors look the same. Anyway the call to a specific testbed connector requires specific parameter entities, so for each testbed a different task is needed. These parameter entities are well known to the system and they are expected to be filled in by the final users, so a task template can be predefined for each testbed connector plugged into the TEFIS System.

By the way, since any common communication protocol has been fixed for the interactions with the testbed connectors, each ProActive task dedicated to the interaction with a specific testbed connector, hides the logic to manage the specific protocol adopted in each case.

Below is reported a Task Template for a connector X.

```
package org.tefis.coreservice.tasks;
public class TaskTemplate_X extends JavaExecutable {

    //   The following fields are automatically set by the default init method
    /* in this case the script for the testbed X is a String containing the script itself,
       but it could be the URI of the actual script for example */
    private String executableEntity_scriptX;
    private int executableEntity_scriptX_Type;
    // another field needed to define the ExecutableEntity of the testbed X
    private int executableEntity_fieldX;
    private String experimentEntity_username;
    private String experimentEntity_inputURI;
    private String experimentEntity_outputURI;
    private String resourceIdentifier;

    public Serializable execute(TaskResult... results) throws Throwable{
        //call the connectorX
    }
 [...]
}
```

The task template is characterized by the specific list of parameters (in terms of parameter name and type) and by the logic of the `execute` method. The execute method wraps the logic to interact with the connector X. Some of this logic should be common to all the connectors, since it is based on the common TCI-EXEC API, while part of it could depend on the parameters specific of each connector (some initialization or pre-processing could be done for example) and on the specific communication protocol.

A task created from a given template will set the parameters with the values given by the user, and will interact with the related connector according to the logic defined in the template's code.

A specific instance of the task is defined for an experiment run, where the task is composed with other tasks to form a whole job. The corresponding task definition in the ProActive job looks like:

```
<task name="Task_on_testbed_X">
        ...
        <javaExecutable class=" org.tefis.coreservice.tasks.TaskTemplate_X">
                <parameters>
                        <parameter name="executableEntity_scriptX" value="..." />
                        <parameter name="executableEntity_scriptX_Type" value=".." />
                        <parameter name="executableEntity_fieldX" value=".." />
                        <parameter name="experimentEntity_username" value=".." />
                        <parameter name="experimentEntity_inputURI" value=".." />
                        <parameter name="experimentEntity_outputURI" value=".." />
                        <parameter name="resourceIdentifier" value=".." />
                </parameters>
        </javaExecutable>
</task>
```

3.3.3. The experiment definition and execution

The final users have to define mainly the scripts and the values of the needed parameters to run the experiment step on each testbed, but not the tasks included in the ProActive job, which are like internal system tasks. The composition and submission of the resulting job on the Scheduler is up to the upper layer building blocks of TEFIS. The main flow of execution is as follows:

- The experiment flow is designed by the user with the support of the Experiment Manager User Interface, made available on the TEFIS portal (cf. [3], section “Experiment Manager Interface”, for more details on the experiment design on the TEFIS Portal);
- The resulting experiment flow is then submitted to the Scheduler by the Experiment Manager component (cf. [4], section Experiment Manager – Other Building blocks”, for more details about this interaction) to be executed;
- the status of the experiment execution may be monitored via the API provided by the Scheduler.

Below is reported a sequence diagram representing this execution flow.

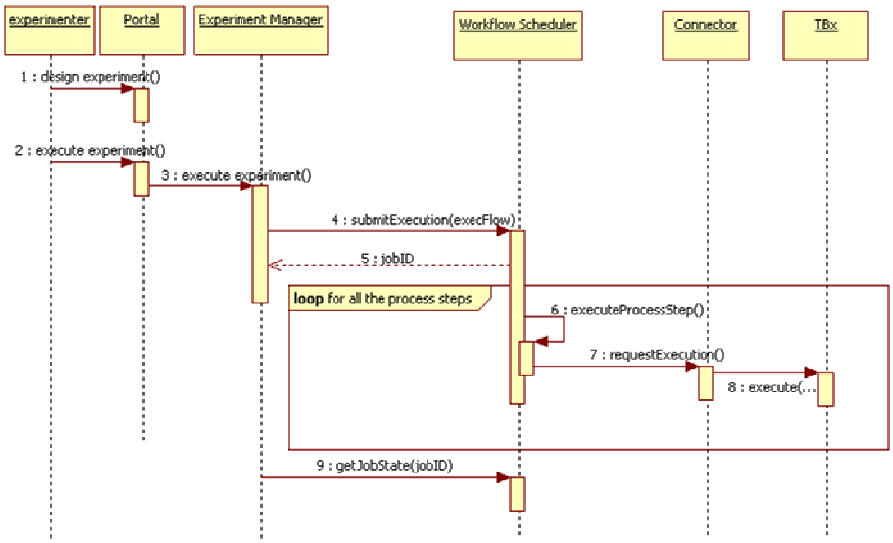

Figure 3 Experiment flow execution

The diagram does not report a complete and detailed representation of the interactions, but it highlights the fact that the experimenter should be able to design the experiment on the Portal (defining a test plan), and then to request its execution (creating a test run and starting its execution) without caring about internal details of the job definition. The experimenter should be able to design the tasks composition in order to run his scripts on each testbed involved in the experiment. He defines in the experiment flow the tasks parameter values expected for a specific testbed involved, but he should not know which one is the class implementing the specific task template he needs. That means that the Portal should present to the users the list of expected parameters, together with their type, for each potential task, i.e. for each task template. In other words, the Portal should know the list of the available task templates, and for each task template it should be able to identify the expected parameters.

This feature could be implemented in many ways, depending on the level of dynamicity preferred (hard coded, configuration files, database, etc.), but conceptually a kind of repository of tasks, each one with the description of the expected parameters, should be available to the Portal building blocks.

3.4. TEFIS specific developments

The TEFIS Scheduler solution, based on the ProActive Scheduling tool, has required several extensions to the ProActive Scheduling itself, and other few pieces of code to realize the whole solution.

The task templates approach described above consists in the developed of several templates needed to support the interaction with a specific testbed. For each testbed a task template is typically needed, even if more then a task should be considered for a testbed. Anyway for the first version of the TEFIS prototype, only one task template per testbed has been developed. The task templates are developments strictly related to the testbed Connector API and they are building blocks external to the ProActive Scheduling tool itself.

An important component that has been realized to support the TEFIS solution has been the new REST API that is exposed by the ProActive Scheduling tool. The REST API allows an easier integration of the ProActive Scheduling tool within the TEFIS platform. The REST API exposes on HTTP protocol the Java API we have seen in the section 3.2.3 and then it allows external components to interact with it in a more

interoperable and loosely coupled way. The Experiment Manager is the TEFIS component who actually interacts with the Scheduler.

Other developments have been required by the first TEFIS solution or have been introduced looking forward at the next TEFIS release. They involve the internal logic of the ProActive Scheduling tool, such as the mechanisms allowing more dynamic scheduling policies. Right now the scheduling process adopts a best effort approach, without allowing the user to define when the execution of a task can occur. A preemptive acquisition of computing resources would allow the scheduling policy to evaluate a user defined property expressing the time when the associated task has to be executed, and then to actually schedule the task at the right moment. This feature is not yet leveraged in the first TEFIS release, but it will be taken in account in the next release, when a TEFIS Scheduling Policy should be introduced.

4. Prototype installation

The TEFIS Scheduler prototype is currently installed in the TEFIS infrastructure operated by INRIA (cf. [9][9] for more details on the TEFIS infrastructure). The TEFSI Scheduler prototype includes the ProActive Scheduling tool, with the appropriate extension to its internal logic, the task templates developed for each testbed, the ProActive Scheduling REST API. They are currently deployed on the Virtual Machine VM2 (tefis2.inria.fr), and the ProActive Scheduling tool leverages, as said, the ProActive Resourcing tool, that is installed in the infrastructure too.

4.1. Scheduler server

The Scheduler server 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. More folders are available in the distribution but these are the ones which are needed to start the server.

To start the server a command shell has to be started into the `bin/[os]` directory, into the installed scheduler home path. To launch the server execute the command

```
scheduler-start-clean[.bat] –u [resource-manager-URL]
```

It will 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, previously started and listening on the given `resource-manager-URL`. Scheduler starting sequence is finished when Scheduler successfully created on `rmi://hostname:port/` is displayed. At this point the ProActive Scheduler is started.

To test the installation a sample job can be submitted, via the Command Line client or via the Graphical User Interface client.

The Command Line tool is included in the distribution. To start it, in the same `bin/[os]` directory, execute de command:

```
scheduler-client[.bat]
```

it will request for login and password (a test login and password are “user” “pwd”). Then, in the console, submit a test job and check its status executing the commands:

```
submit(“../../samples/jobs_descriptors/Job_8_tasks.xml”)
jobstate([job id received before])
```

The full list of command available in the console is shown executing in the console:

```
help() (or ?)
```

The job submission and the Scheduler activity can be also seen via 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.

The current installation is available at:

```
rmi://tefis2.inria.fr:1099/
```

4.2. Scheduler REST API

The Scheduler REST API is a Web application to be hosted in a J2EE container. In the current installation on the TEFIS Infrastructure it is hosted in the Apache Tomcat 6.0 server.

It is distributed as a `war` archive. The `war` file contains a property file named `portal.properties` that contains the following configuration properties:

- `scheduler.url` = the URL of the Scheduler
- `rm.url` = the URL of the Resource Manager
- `scheduler.cache.login` = the login of the account to use to cache the scheduler state (it must have the right to download the scheduler state)
- `scheduler.cache.password` = password linked to the 'cache' user.

Once the archive is deployed in the server (start the server if it is not yet running), the REST API is available at

```
http://hostname:port/proactive_sched_rest/scheduler
```

To test the installation the following command lines could be used (from a Linux machine):

- Login

```
curl -d "username=demo&password=demo"
http://hostname:port/proactive_sched_rest/scheduler/login
```

- submit a job

```
curl -H "sessionid:1" -F 'file=<Job_2_tasks.xml'
http://hostname:port/proactive_sched_rest/scheduler/submit
```

-list jobs (pending, running, finished)

```
curl -H "sessionid:1" -H "Accept: application/json"
http://hostname:port/proactive_sched_rest/scheduler/jobs/
```

-list job 10's tasks

```
curl -H "sessionid:1" -H "Accept: application/json"
http://hostname:port/proactive_sched_rest/scheduler/jobs/12/tasks
```

-delete a job

```
curl -X DELETE -H "sessionid:1" -H "Accept: application/json"
http://hostname:port/proactive_sched_rest/scheduler/jobs/10
```

-result from a task

```
curl -H "sessionid:1" -H "Accept: application/json"
http://hostname:port/proactive_sched_rest/scheduler/jobs/1/tasks/task1/result
```

The current installation is available at:

```
http://tefis2.inria.fr:8080/proactive_sched_rest/scheduler
```

5. Conclusion

In this document we have analyzed the requirements and features an eventual existing component must match to be used as the Experimental and Workflow Scheduler in the TEFIS platform. We stated that the ProActive Parallel Suite and in particular the ProActive Scheduling tool contained in that suite provides most of the required features. Concerning the not matched requirements we explained how we have extended the ProActive Scheduling tool to cover those missing functionalities. Those extensions are related to two particular aspects: *Task Template* and *REST API*.

To validate the proposed solution we described how the ProActive Scheduling tool enacts the experiment defined by the user. We investigate, especially, the interaction of the ProActive tasks and the testbed connectors presenting the need and the advantage offered by the use of the Task Template in such interaction.

The last part of the document is dedicated to the description of the prototype carried out and how it is installed on the INRIA machines. We present some of the most useful commands to do job and task management in the ProActive Scheduling tool too using the command line and the REST API.

References

[1] D2.1.1: Global architecture and overall design

[2] D4.1.1_Resource_Management_implementation_draft.doc

[3] D3 3 User tools implementation.doc

[4] D3 2 Building blocks integrated into TEFIS portal.doc

[5] D3.1.1_Teagle_assessment_and_TEFIS_portal_specifications

[6] http://proactive.inria.fr/release-doc/Scheduling/single_html/ProActiveSchedulerManual.html

[7] D5.1.1 - Generic and Specific Connector specifications v1.0.doc

[8] http://www.activeeon.com/public_content/schemas/proactive/jobdescriptor/3.0/schedulerjob.xsd

[9] D2.3.1 – Integration Plan.doc