Workflow application development with Web-Services on EGEE

Tristan Glatard

June 26, 2007

1 Introduction (10 minutes presentation)

The goal of this tutorial is to present workflow development systems and to interface them with the EGEE grid through Web-Services. Sections 2 and 3 will be devoted to workflows description with Taverna[1] [2] and to their efficient execution with MOTEUR[2] [1]. Then, Section 4 will propose Web-Service development and deployment with gSOAP[3] and Apache httpd web server and Section 5 will finally interface Web-Services to the EGEE grid. The main concepts used in this practical are briefly introduced here.

1.1 Workflow models

Workflow approaches can be categorized in 5 broad classes with respect to the amount of information (concerning functions, data and resources) contained by the workflow description:

- Functional workflows only contain a definition of the functions to be executed by the workflow
- Task graphs contain an instantiation of the function on the data
- Service workflows contain a location of the functions on resources
- Executable workflows contain functions, resources and data instantiation.

The goal of a workflow engine is to have the workflow move to an executable representation. Relations between those classes are depicted on figure 1. In this tutorial, we will focus on services workflows.

1.2 Parallelism in a service workflow

Three kind of parallelism may be exploited in a service workflow: workflow parallelism, data parallelism and service parallelism. Workflow parallelism is the most simple one: it corresponds to the concurrent execution of two independent services of the workflow on two independent data items. Data parallelism corresponds to the concurrent processing of independent data items by a single service. Finally, service parallelism (also denoted pipelining) is the concurrent execution of two independent data items by two services linked by a precedence constraint.

[1] http://taverna.souceforge.net
[2] http://egee1.unice.fr/MOTEUR
[3] http://www.cs.fsu.edu/~engelen/soap.html

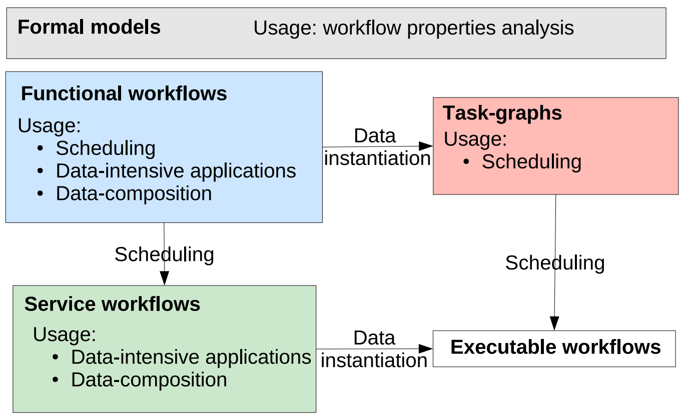

Figure 1: Classification of workflow approaches

1.3 Web-Services

The goal of Web-Services is to provide *platform-independence*, *dynamic location* and *loosely coupling*. A Web-Service is described by a Web-Service Description Language (WSDL) file that specifies two kinds of XML tags:

Tags describing *what* to invoke:.

- *Type*: the types of the exchanged data
- *Message*: a set of *parts*: a part is an input or output parameter of the service and a message is a service request or response
- *Operation*: a set of messages: the equivalent of a method in OO programming
- *Port type*: a set of operations: the equivalent of a class in OO languages

Tags describing *how* to invoke it:.

- *Binding*: the protocol used to invoke the service (*e.g*: SOAP/HTTP)
- *Port*: the endpoint of the service (*i.e* an Internet address and port)
- *Service*: a set of ports

1.4 General remarks

A indicative time-line is provided for each section and sub-section of this tutorial. Don't hesitate to skip some paragraphs and/or spend more time on other ones according to your experience and interests.

Tools and their dependencies to download will be indicated along the practicals. Most of them are also available from http://colors.unice.fr/tutorialBudapest/sources for convenience.

The practical part of this tutorial has been tested on a Linux Fedora Core 7 without root access. Some details may change if you are running another Linux distribution.

☜ Practical sections are identified by this symbol.

2 Workflow prototyping from Beanshells with Taverna (65 min)

Taverna is a workflow manager developed inside the myGrid UK e-Science project. It is based on the Scufl workflow language. In the Scufl language, components of the workflow are called *processors*. The kind of processors that will be used in this practical are Web-Services, Beanshells[4] and String constants. Processors have input and output *ports* (*i.e* parameters) that may be connected with *data links*. An output port may be connected to several input ports. Similarly, an input port may be connected to several output ones. Inputs and outputs of the whole workflow may also be defined. The goal of this Section is to create a simple workflow with the Taverna workflow manager.

> ☜ **Taverna installation (10 min).** On your local machine, download the Taverna 1.5.2 archive[5] and make sure that java 6[6] (or at least 5) is in your `PATH` (if not, add it with `export PATH=javadirectory:$PATH`) and launch Taverna by running the `runme.sh` script.

Before submitting real grid tasks (in Section 5), we are first going to describe a simple workflow from Beanshells, that are components that allow the user to write some lines of Java directly inside a processor. To be more realistic, the processing time of the Beanshell will be simulated by `sleep` commands.

> ☜ **Simple workflow description (15 min).** Add a Beanshell processor to the workflow: on the upper-left pane of the Taverna workbench, browse the `Local Services` list, right-click on the `Beanshell scripting host` and add it to the workflow. In the advanced model explorer (bottom-left pane), right click on the new Beanshell processor and choose `Configure Beanshell`. In the `Ports` tab, define two inputs: `sleepTime` and `message` as Plain text values and a `result` output (Plain text value also). If you configure the diagram accordingly, those ports will appear on the right pane of the workbench. Add the following Java code to the Beanshell by copying it in the `Script` tab:

```
int i = Integer.parseInt(sleepTime);
Thread.sleep(i*1000);
result="I slept "+sleepTime+ "s and then I printed \""+message+"\"";
```

> Therefore, the executing time of the resulting Beanshell is determined by its `sleepTime` input.
>
> In the bottom-left pane of the workbench, add two inputs to the workflow and connect them to the ports of the Beanshell by right clicking on them and choosing the suitable destination port. The corresponding data link appears in the workflow description. Add an output to the workflow and connect the `result` output port of the Beanshell to it. The resulting workflow is pictured on figure 2. Save the workflow and have a look at the generated Scufl document: processors are first described, and then data links are specified. The corresponding Scufl document can also be downloaded[7].

[4] http://www.beanshell.org
[5] http://taverna.sourceforge.net
[6] http://developers.sun.com/downloads
[7] http://colors.unice.fr/tutorialBudapest/workflow1.xml

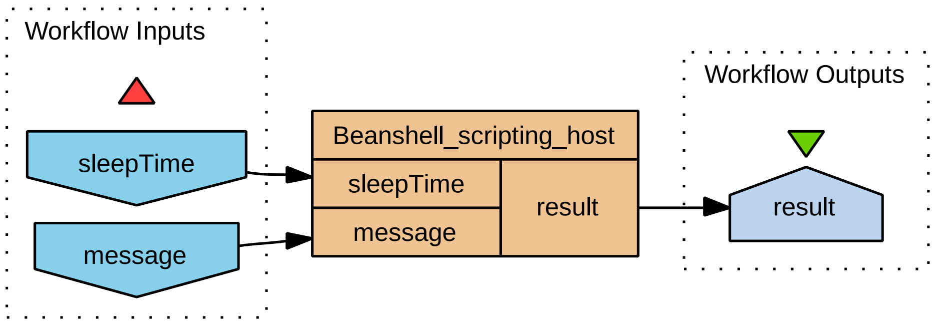

Figure 2: Simple workflow with a `Beanshell`

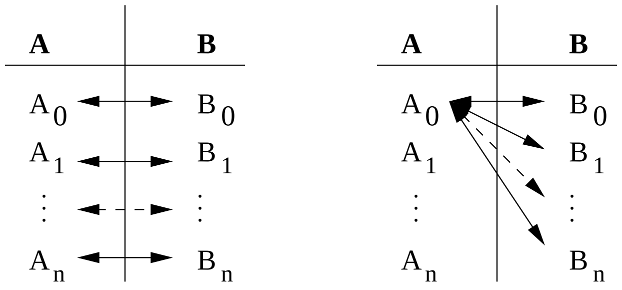

Figure 3: Action of the dot (left) and cross (right) products

At a given instant, the input ports of a processor may contain several data items and the processor has to be iterated on them. In Scufl, the iteration pattern of a processor on its input ports is specified by *iteration strategies*. The default iteration strategies is a `cross product`. The alternate iteration strategy is the `dot product`. The action of cross and dot products is depicted on figure 3

☜ **Workflow enactment and iteration strategies (10 min).** Click on the `Results` button on the top of the workbench and specify inputs for the workflow: add two different data items (`10` and `30` to have significantly different execution times) for the `sleepTime` input and a single text message for the `message` input. Finally, click on the `Run workflow` button to launch the execution. The Beanshell processor is invoked twice, once for each value of `sleepTime`.

Launch a new execution with two values in the `message` input. Now, the Beanshell processor is invoked 4 times, once for every `message/sleepTime` pair. Indeed, the default iteration strategy is a *cross product*. Come back to the `Design` pane and select the Beanshell processor. In the `Metadata for 'X'` tab, click on `Create iteration strategy` and change the cross product to a dot. Then, run the workflow again: the Beanshell processor is invoked only twice.

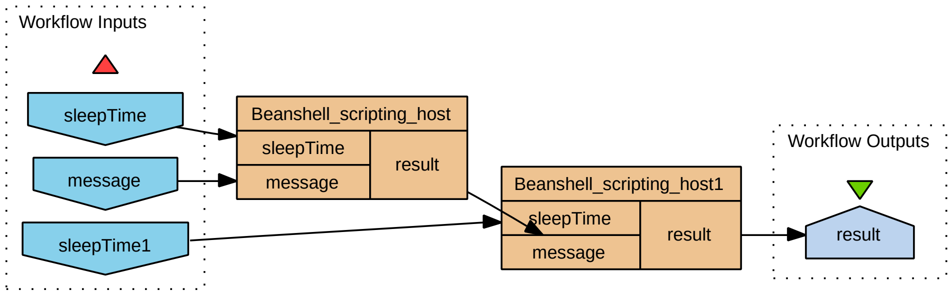

Figure 4: Another `Beanshell` workflow

☜ **Enabling data parallelism (5 min).** You may have noticed that the Beanshell processor has been invoked *sequentially* on the 4 (or 2, depending on the iteration strategy) data items. This is clearly not suitable for a grid execution. To avoid that, select the Beanshell processor in the bottom-left pane of the workbench and increase its number of threads. It enables data parallelism for this processor.

Apart from this data parallelism, workflow and service parallelism are also required for an efficient execution. The goal of the next paragraph is to highlight them.

☜ **Service parallelism (10 min).** Add another sleeping Beanshell processor `Beanshell1` with the same inputs/output and Java code as the previous one and with (for instance) 10 threads. Add a new input `sleepTime1` and connect it to the `sleepTime` input of `Beanshell1`. Connect the `result` output of `Beanshell` to `Beanshell1` in order to create a data link between those two Beanshells. Remove the link between `Beanshell` and the output of the workflow and create a new one between `Beanshell1` and the output. Set the iteration strategy of `Beanshell1` to a dot product. This workflow is depicted on figure 4. Execute this workflow with a single `message` input, 10 and 30 in the `sleepTime` input and 30 and 10 in `sleepTime1`. Each Beanshell runs twice. However, the execution is not pipelined (*i.e* service parallelism is not present) and the execution is close to 1 minute, whereas it could be 40 seconds.

In order to precisely measure the duration of the workflow, we are going to use *coordination constraints*, that allow to specify precedence constraints among the processors, even in absence of data links.

☜ **Workflow duration measurement (15 min).** Add a `getStartDate` Beanshell processor with no input, 1 `startDate` output and the following Java code:

```
startDate=System.currentTimeMillis();
```

Similarly, add a `getStopDate` Beanshell processor with no input, 1 `stopDate` output and the following Java code:

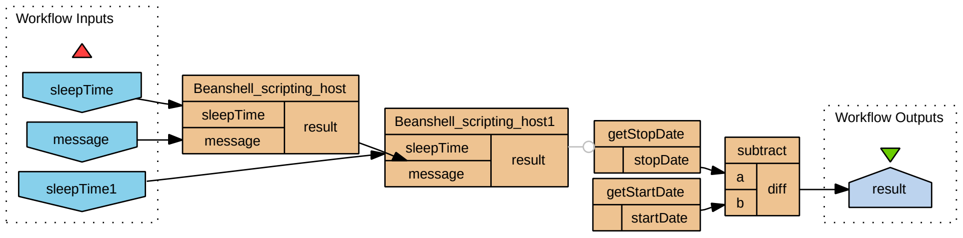

Figure 5: Workflow that measures its execution time thanks to a coordination constraint.

```
stopDate=System.currentTimeMillis();
```

Add a coordination constraint to the `getStopDate` processor in order to make it start after the `Beanshell1` processor: right-click on `getStopDate` on the bottom-left pane of Taverna and choose `Coordinate from Beanshell1`. Finally, create a `subtract` Beanshell processor with 2 (Plain text) inputs `a` and `b` and a single Plain text output `diff` with the following code:

```
diff=Long.parseLong(a)-Long.parseLong(b);
```

Add the required data links in order to have `subtract` compute the execution time of the workflow. The subsequent workflow is depicted on figure 5. It is also available for download[8]. Run this workflow with the same input data set than before (single `message` input, 10 and 30 in the `sleepTime` input and 30 and 10 in `sleepTime1`) and check that the execution time is close to 1 minute.

3 Service parallel execution with MOTEUR (30 min)

The goal of the MOTEUR workflow manager is to enable service parallelism (*i.e* pipelining) in the execution of Scufl workflows in addition to data and workflow parallelism that are already present in Taverna. Handling *dot* products in a service and data parallel workflow is not straightforward because data items are completely puzzled during the execution. The solution adopted in MOTEUR is to consider a graph whose nodes are the data items produced by the execution of the workflow and whose directed edges mean "produces". The user defines group among the inputs of the workflow and two data items will be combined in a dot product if and only if the ancestor of both of them in the group have the same rank in the original sources of the workflow.

☞ **MOTEUR installation (15 min).** Download the current MOTEUR version[9] and its dependencies:

- Apache Axis 1.4[10]
- The Beanshell jar[11]

[8]http://colors.unice.fr/tutorialBudapest/workflow2.xml
[9]http://egee1.unice.fr/MOTEUR
[10]http://apache.miroir-francais.fr/ws/axis/1_4/
[11]http://www.beanshell.org

- The graphviz `dot` graph rendering tool[12]

Set the `JAVA_HOME` variable to the java installation directory and the `MOTEUR` variable to the MOTEUR directory. Add all the Axis' jars to your `CLASSPATH` with the following loop:

```
for i in /home/.../axis-1_4/lib/*.jar; do
export CLASSPATH=$CLASSPATH:$i; done
```

Add the Beanshell jar (`bsh-2.0b4.jar`) to your `CLASSPATH` as well. Check that the `dot` graphviz tool is in your `PATH`. Finally, compile MOTEUR with `make`.

In MOTEUR, the input data of the workflow is described in a dedicated XML file. Each input of the workflow is associated to a `<source>` XML tag and the items to be put in this input are defined by the `<item>` tag. Groups between sources may be defined by the `tag` attribute of the `source` tag.

☜ **MOTEUR workflow execution (15 min).**

In the MOTEUR directory, launch MOTEUR with `java engine.Moteur`. Open the last workflow created in Section 2 and add the input data file[13]. This file contains the same input data set than the one used in the previous section. A group is defined between sources `sleepTime` and `sleepTime1` to have a correct semantic of the dot product of the `Beanshell1` processor.

Launch the workflow by clicking on the `Run` button (ALT-R). The display can be refreshed on demand with the `Refresh` button (ALT-e).

The `Services` tab can be used to get some information about the services of the workflow. In particular, time statistics are available for each service. A list of produced data items is also available in this tab. The data tree can be browsed in order to see the provenance of the data.

Service parallelism is enabled in MOTEUR: the execution of this workflow should be close to 40 seconds. At the end of the execution, the corresponding result is available in the `Results` tab.

4 Web-Service development with gSOAP (40 min)

Thanks to the wide adoption of the standard, Web-Services facilitate code reusability and interoperability. They also allow distant execution of the components of the workflow, which was hardly possible with Beanshells. We will rely on the gSOAP toolkit for Web-Service development.

☜ **gSOAP installation.** Download the gSOAP binary version[14]. Set the `GSOAP` variable to the gSOAP directory and check that `soapcpp2` is in your `PATH`.

[12] http://www.graphviz.org/

[13] http://colors.unice.fr/tutorialBudapest.inputs.xml

[14] http://www.cs.fsu.edu/~engelen/soap.html

gSOAP is a toolbox that allow the creation of Web-Services in C and C++. It provides a compiler (`soapcpp2`) that is able to generate WSDL files and associated stubs from a `.h` header. A WSDL parser (`wsdl2h`) performs the reverse operation. We are going to use gSOAP to replace the `subtract` processor of the previous workflow by a Web-Service. Actually, such a subtraction service is available as a sample in the gSOAP distribution and we can use it without writing a single line of code.

> **Standalone deployment of the `calc` sample (10 min).** Compile the `calc` sample of the gSOAP distribution with `make` in the `/home/.../gsoap/samples/calc` directory. Edit the generated WSDL file and change the location to `http://localhost:25000` so that your Web-Service can be deployed locally. It can also be done prior to the compilation by editing the `calc.h` file. Then, launch `calc_server` on port `25000` (`./calc_server 25000`). In the top-left pane of Taverna, right-click on the `Available processors` and add a new WSDL scavenger. Specifies the local path to the WSDL (`file:///home/.../calc.wsdl`). The 5 different *operations* of the `calc` *service* are now available in the workbench. Add a new processor from the `sub` operation and replace the former `subtract` by this Web-Service one. Check the compatibility by running this Web-Service workflow with MOTEUR. The Scufl document of this workflow is also available for download[15].

This standalone deployment of Web-Services with gSOAP is very convenient as it does not require any web server installation nor configuration. Actually, gSOAP embeds a lightweight web server in its services, to make them able to handle SOAP requests. However, there might be some scenarios for which this standalone deployment is not suitable. For instance, if you plan to deploy several services on the same machine, then you would not be able to use port 80 for all the services and the other ports may not pass through firewalls.

gSOAP provides two alternate ways to deploy a Web-Service: the CGI and an Apache module. Both of them rely on the Apache `httpd` Web-Server. In this practical, we will test the CGI deployment.

> **Httpd installation (10 min).** Check whether `httpd` is installed and running on your machine (for instance with `/sbin/chkconfig --list`) and if you have write access to the web repository. If one of those conditions is not fulfilled (most probable), then download the `httpd` web server version 2.2[16]. Configure it to be installed on a local dir with `./configure --prefix=/home/.../`, compile and install it with `make; make install`. The main configuration file of the server is `conf/httpd.conf`. Among several other parameters, you can configure the port on which it is listening (`Listen` directive), the path to the web directory (*i.e* the directory from which documents will be served) and allow the execution of CGI scripts on some directories. By default, the server is running on port 80 and CGI-scripts are allowed only in the `CGI-bin` dir. If you are not `root` on the machine, change the listening port to `8080` and launch the web server with `bin/apachectl -k start`. Point a browser to `http://localhost:8080`: a web page should appear if everything works fine.

[15] `http://colors.unice.fr/tutorialBudapest/workflow3.xml`
[16] `http://httpd.apache.org`

The Common Gateway Interface (CGI) is a simple standard allowing a web-server to call an executable in order to produce dynamic web pages. In the context of this tutorial, the CGI script will produce some SOAP XML code.

> ☞ **CGI deployment of the Web-Service (10 min).** To deploy the `calc` Web-Service as a CGI in your `httpd` server, just copy the `calc_server` binary to the `cgi-bin` directory. Don't forget to change the location [17] in the `WSDL` document of the service. Make sure that the workflow still correctly runs with Taverna and MOTEUR.

> ☞ **Your own example (10 min).** Following the example provided by the `calc` sample, replace the `Beanshell1` processor of the workflow of Section 2 by a Web-Service. To do that, first write a `.h` file containing the prototype of the web method as well as some information used by the compiler to generate the WSDL (look at `calc.h` for an example). Then, write the implementation of your web server in a `.c` file: the main method will be strictly identical to the one of the `calc` example and you only have to write the implementation of the web method. The binary needs to be linked with `soapcpp2.o` which can be produced from the `soapcpp2.cpp` source file from the gSOAP distribution (`g++ -c soapcpp2.c`).

5 Interfacing Web-Services with EGEE (40 min)

The Grid Application Service Wrapper (GASW) aims at interfacing application Web-Services with the EGEE grid. Given the description of a command-line, it provides some functionalities to generate a correct JDL and to transfer all the required files to the worker node, eventually via the UI. Once deployed and configured, an application can be accessed transparently as every Web-Service. In this practical, the GASW server will run on your local host and connect to the UI through ssh.

> ☞ **UI configuration and ssh access (5min).** Initialize a valid proxy on the UI with `voms-proxy-init --voms gilda`. The GASW service will need a non-interactive `ssh` access to the UI. To do that, create a public/private dsa key pair with `ssh-keygen -t dsa` (or use your pair if you already have one).
>
> On the UI, create a `.ssh` directory on your homedir and copy your public key (`id_dsa.pub`) in a `authorized_keys` file in the `.ssh` directory. Set the permissions of the `.ssh` directory to 700 and the ones of the `authorized_keys` file to 600.
>
> On your local host, launch an `ssh-agent` and execute in a console the output lines (export of some variables). Add your private key to the agent with `ssh-add /.../id_dsa` and type the passphrase. Check that your can access the UI from your host with `ssh budapestXX@glite-tutor.ct.infn.it` without typing any passphrase.

The Grid Application Service Wrapper (GASW) is available with the MOTEUR code.

[17] http://localhost:8080/cgi-bin/calcserver

> **GASW installation (5 min)**. First, set your `GSOAP` variable to the gSOAP directory and the `MOTEUR` variable to the MOTEUR directory. Then, in the MOTEUR directory, compile `libGrid` and `libGasw` with `make grid`. Those library are the core of the GASW service. They must be added to the library path before starting it: `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MOTEUR/.libs`. The Web-Service interface is based on gSOAP and located in the `gaswService/WS` directory. Type `make` in this location to compile the service. It produces `gasw_server` and `gasw_client`.

The GASW server is able to execute any command line on EGEE, given a simple description. An example of such a description is available[18]. This description corresponds to the execution of a `hello` binary which is available from the `colors.unice.fr` web server. It takes as input a string with option `-f` and outputs a grid file (LFN) with option `-o`. It requires an additional file (`header.txt`) which is also available from a URL. Such additionally (sandbox) files could for instance be external dependencies such as dynamic libraries.

> **A simple example (10 min).** To run properly, the `gasw_server` requires a grid configuration file which is pointed by the `GRIDCONF` variable. An example of such a file is in `MOTEUR/conf/grid.conf`. It specifies the login to be used to log on the user interface (here: `budapest{1..40}`), the name of the user interface (here: `glite-tutor.ct.infn.it`), the path to the ssh key to be used to log in, the type of grid (here: `LCG`), the virtual organization (here: `gilda`), the storage element to be used to store files (choose on from `lcg-infosites --vo gilda se`), the timeout value to be assigned to jobs (in seconds), the retry count and some environment variables to be set to the job on the worker node (in this case, setting the `LFC_HOME` to `$LFC_HOST:/grid/gilda` may be necessary). Once the grid file is correct and the `GRIDCONF` variable points to it, the server can be launched as a standalone gSOAP Web-Service
>
> ```
> ./gasw_server 18000
> ```
>
> The client can then be executed by:
>
> ```
> ./gasw_client http://localhost:18000 http://colors.unice.fr/hello.xml "foo"
> ```
>
> The first argument of the client is the endpoint of the Web-Service, the second one is the URL of the command-line description file and the last one is a string containing the arguments of the command-line.
>
> The server should submit and monitor a job that correspond to the correct command-line. Output file names are uniquely generated by the GASW and are terminated by a `.gen` extension by default. At the end of the execution, check that the produced LFN exists and has the correct content.

Given an appropriate XML description, this server should be able to execute any command-line on the grid. However, one should be aware of the potential security risk that would be taken by an uncontrolled deployment of such a service. Thus, this service must be deployed on a web server handling authentication and authorization.

As a Web-Service, the GASW can be added to any Scufl workflow.

[18] `http://colors.unice.fr/hello.xml`

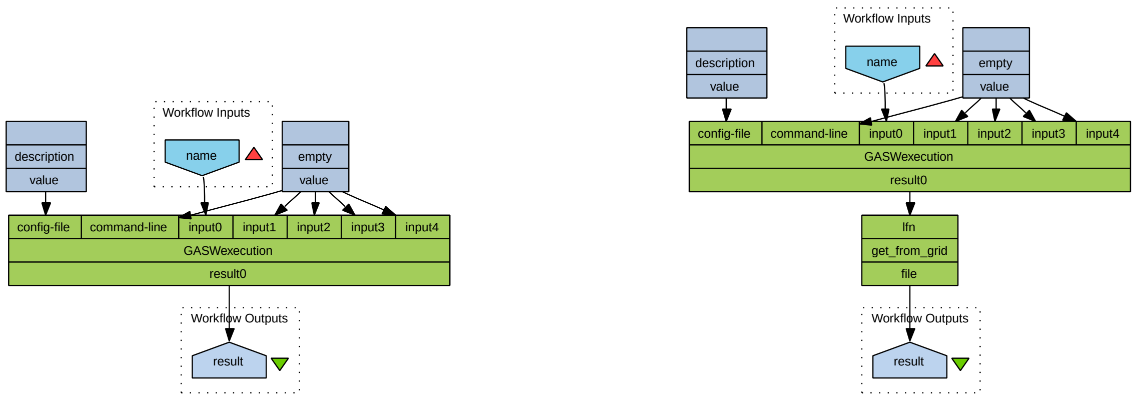

Figure 6: Left: EGEE-enabled workflow; Right: with data transfers

☞ **A grid-enabled workflow (10 min).** The WSDL document of the GASW is available from `http://colors.unice.fr/gasw_service.wsdl`. It specifies that the endpoint of the server is located at `localhost:18000`. Include this WSDL in Taverna and add the `GASWexecution` operation to a new workflow. This operation has 7 inputs: the description file of the command-line to be executed, the command-line itself and 5 potential inputs.

Right-click on the `String constant` processor in the `Local services` list of the top-left pane of the Taverna workbench and add a string constant containing the path to the `hello` description file to the workflow and connect it to the `config-file` input of the GASW. Similarly, create a `String constant` containing an empty string (*i.e* "") and connect it to the `command-line input` of the GASW. Add a new input to the workflow and connect it to the `input0` of the GASW. `input1` to `input4` will also be connected to the empty string as only a single input is required for the service. Add an output to the workflow and connect it to the `result0` output of the GASW. `result1` to `result4` will not be used because the `hello` executable only returns a single output. The resulting workflow is depicted on the left of figure 6 and available for download[19] as well as an input file[20]. Run it with MOTEUR and check that the file has been correctly produced and stored on the specified SE.

Data transfers are not handler either by Taverna or by MOTEUR. To handle them and make the results of the workflow available, a dedicated service has to be integrated inside the workflow. Such a service is also available with the MOTEUR code. Similarly to the GASW, it connects to the UI through ssh and execute the appropriate `lcg-cp` command.

☞ **Handling data transfers (10 min).** The data transfers service is a Web-Service based on gSOAP. It can be compiled by typing `make` on the `data_transfers` directory of the MOTEUR code. The WSDL of this service is available here[21] and the endpoint specified by this document is `localhost:19000`.

[19]`http://colors.unice.fr/tutorialBudapest/workflow4.xml`
[20]`http://colors.unice.fr/tutorialBudapest/inputs2.xml`
[21]`http://colors.unice.fr/data_transfers_service.wsdl`

This service downloads an LFN from EGEE to a local repository (`MOTEUR/data`). Make sure that this repository exists (create it if not) and add the data transfers service to the EGEE workflow, in order to obtain the workflow depicted on the right of figure 6. This workflow is also available for download[22]. Start `data_transfers_server` on port 19000 and run the workflow. Once the workflow has finished, your results are available in the `Results` tab of MOTEUR. Following the links should allow you to visualize the produced file by clicking on the correspond icon.

References

[1] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR. *International Journal of High Performance Computing and Applications (IJHPCA)*, 2007. to appear.

[2] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics journal*, 17(20):3045–3054, 2004.

[22] http://colors.unice.fr/workflow5.xml