jGASW HOW-TO

Authors:
Javier Rojas Balderrama, <javier@i3s.unice.fr>
Johan Montagnat, <johan@i3s.unice.fr>

Table of Contents

Creation of services ... 1
Service wrapping ... 1
User interface ... 2
Tools re-location ... 4
Data management ... 5
Execution ... 5
Results manipulation ... 5
Invocation client interface ... 6
Use-case ... 6
Conclusions ... 7

Usually, tools developed in the scientific community are neither designed for integration in a standard environment nor for tool re-location and remote invocation. The jGASW toolbox aims at providing a wrapper shell for applications using a command-line interface (CLI), using Web Service (WS) standard invocation in order to enable the tools interoperability and dynamic re-location.

jGASW provides much more than a mere invocation interface to the service. This tool provides a complete mechanism to package tools and their dependencies into an archive, relocate this archive on a given server, deploy and publish it. As a submission interface, jGASW is also involved with security (access control to the service execution) and grid invocation (including files transfer for proper processing on a remote resource). jGASW therefore provides a full range of functionality, making the applications autonomous, re-locatable and grid-compliant. It also addresses grid execution performance. The jGASW framework is composed by three elements: (1) a service wrapper, (2) libraries to manage the relocation of resources, and (3) a client programmatic interface to invoke the wrapped applications. The wrapper and the libraries are based on a data modeling describing CLI applications and the way to execute them. In contrast, the client interface is an autonomous service consumer.

This document presents explains the process of building services from CLI application, including the user interfaces; it also shows and overview of the service re-location and execution; finally it describes the access to the Web services and their invocation. In the real example is presented to show the use of jGASW.

Creation of services

Service wrapping

The process to expose an application by way of Web services begins with the generation and compilation of personalized code that reflects the resource description and configuration files to deploy the service on the services container. The description of a CLI application detailed on the User interface section abstracts a application. Now the principle consists to transform that description into a WS interface. In jGASW this transformation is based in a template engine that provides source code and the necessary files to let the container interpret it.

In terms of solutions jGASW uses Sun Metro that provides the stack engine to publish services. It is easily integrated with the Apache Tomcat servlet server and supports additional needs attachments manipulation. The Tomcat server is not configured to support Metro out of the box. Metro should be installed along side Tomcat before hosting any service. Additionally the set of jGASW libraries has to

be accessible in the class path. These libraries implements all the logic associated to the relocation described.

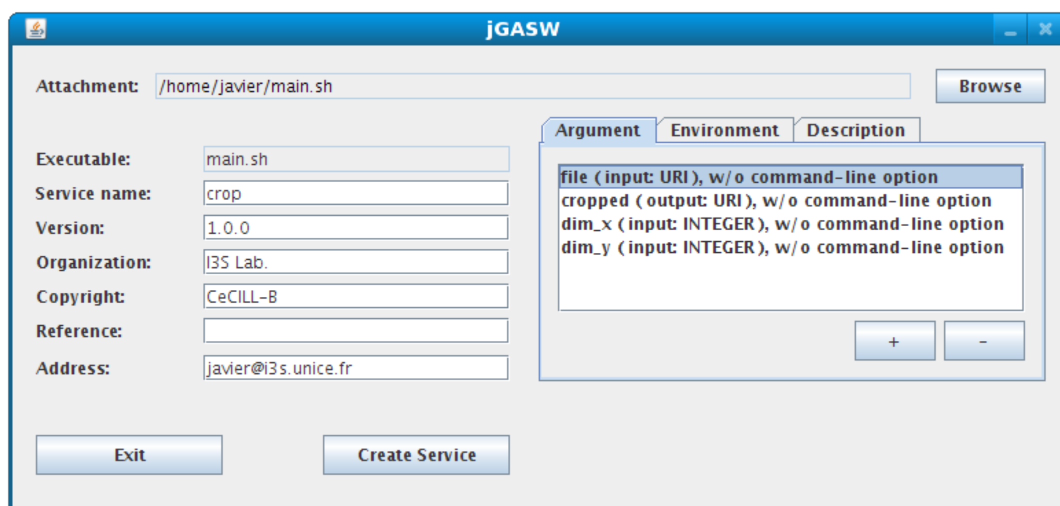
Following the Apache Tomcat server schema, all services are deployed in form of a Web Application Archive (WAR),a special JAR file used to distribute a standard Web application. In the case of jGASW this archive includes configuration files (sun-jaxws.xml and web.xml); the description of the resource (description.jgasw); the wrapped CLI application; the dependencies of the application and the WS interface compiled code. This WAR is created automatically using one of the jGASW interfaces.

The last action to publish a service is the deployment on the container, Tomcat sets up services at run-time without perturbing its normal operation (i.e. there is no need to restart the server). This quality is known as hot deployment. Just after the deployment, the WSDL describing the service is available and ready for invocation. Removing the deployed services releases safely the reference to the service from the container.

User interface

The user interfaces of jGASW allows us to create and deploy new services. The procedure aims at being as simple as possible, filling the description form that includes eventual dependencies of the application and the details of all arguments. This section describes the graphical interfaces for creating and deploying services.

Figure 1. jGASW service creation frame



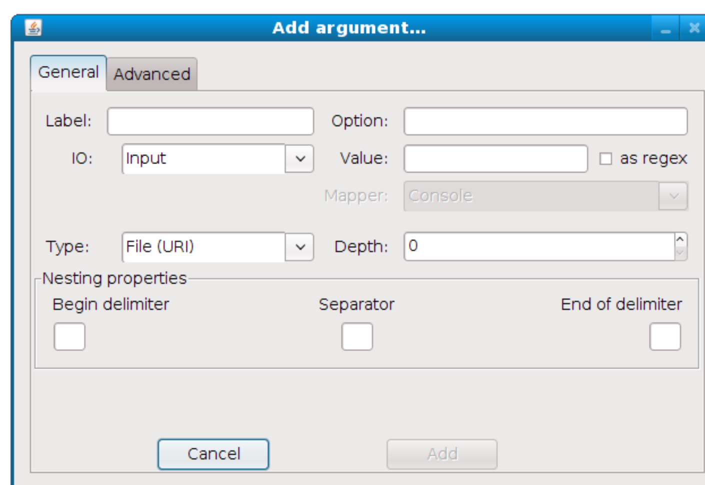
The main window shown on the Figure 1 includes all relevant fields of the CLI application. All these elements are used to generate the description. Below there is a brief description of each element of this form.

- Attachment is the reference path of the application or the ZIP file containing the application and dependencies (loaded using the Load button). If a ZIP file is provided then an auxiliary dialog showing the list of files will appear and the user should select the target file representing the main executable of the service.
- Executable is a read-only field that shows the name of the target file representing the main executable of the service. It typically is as same as the name of the attachment if only a simple file is provided otherwise it is the selected element from the list of ZIP file.
- Service name is the symbolic name of the service.
- Version is the declared version of the service.
- Organization is the name of the organization who owns the application.

- Copyright is the licence category of the original application.
- Reference is a reference or key name associated to the application.
- Address is the contact email address of the person in charge of the wrapped service.

The description, environment and arguments associated to the application in the description are included in tab panes. Once the add button, is clicked the dialog is shown to include either the argument or the environment information. For the case of the application description, it is included in a text area as any other informative field of the reference. Even if the later is purely informative it should describes a detailed description of the CLI application including a command-line example.

Figure 2. jGASW Argument dialog



An argument can be described using the fields shown on the Figure 2 and detailed as follows:

- Label the reference's name of the argument. This label is just a reference freely named by the user. If the Type is declared as Input this label is used as argument name of the Web service.
- Option the command-line option associated to the argument. For instance the posix-style defines options with simple dash character followed by a letter or double dash followed by the option name.
- Value is the exact content of the argument. The value is only taken into account when the IO is not declared as Input because input values are provided invoking the service. The value can contain a case sensitive string or a regular expression. A second alternative is designed for output arguments with variable number of entries in the Advanced tab pane.
- IO the argument's type, namely: Input, Output or Constant. The last type can be used to provide values of arguments that are not inputs nor outputs.
- Type the primitive type class associated to the argument. The possible values are string, double, integer or File (URI).
- Depth the associated value to the dimension of the argument. The default value is zero corresponding to an scalar according to the array programming principles.
- Nesting properties. All elements needed to declare the argument as a array of a given dimension are declared on the nesting frame. For that, the begin and end delimiters (blank space by default) and the separator character are set.

Additionally, advanced argument options may be set on the Advanced pane of the Arguments dialog. These options add richness to the classical description of an argument and is provided for special situations as implicit arguments and special file types. The advanced options includes:

- File Type assumes as default that an argument of type URI is a simple Regular file. The behaviour is modified using the Expand, Replace or Directory options. Expand and Replace are used in combination with a list of extensions. When Expand is selected the argument declared with a specific Value is concatenated to each extension and then add all the combinations to the command-line argument. If the Replace option is set the argument declared with the Value is used in the command-line but the reference a this file is resolved using all the extensions. Finally when the option Directory is selected the declared Value is considered as a directory file name and all files included inside are associated to this argument.

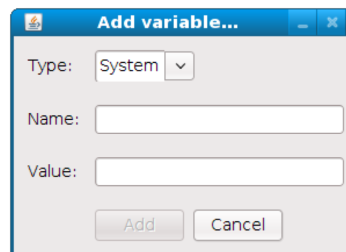
- Extensions is used to declare all file extensions associated to the declared Value in combination with the File Type.

- Space is set when an argument Value is declared next to the Option without any space between them.

- Implicit is set when the described argument is not part of the command-line but this information is necessary to the execution. For example, if files not described in the command-line are needed those can be expressed as implicit values.

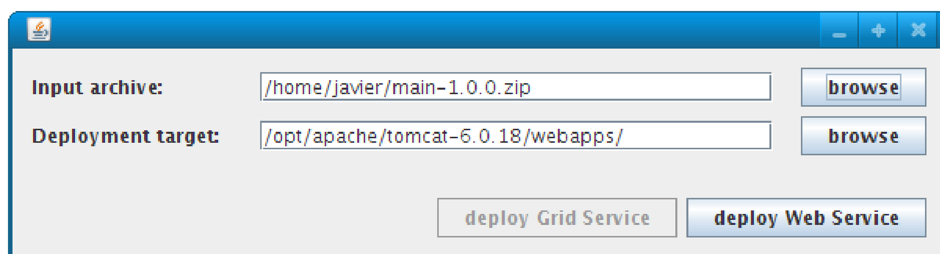
Like arguments, we can also include environment variables. this variables are configuration properties of the application that not are part of the command-line but they are used during runtime. Three categories of variables are pre-defines. System, related with the OS, Grid related with the distributed infrastructure where the application can be deployed and the jGASW properties used for the framework. To achieve variables inclusion click the add button on the environment tab pane. A simple dialog is displayed as shown in Figure 3.

Figure 3. jGASW Variable dialog



Finally to create the ZIP file representing the service once the description, arguments and environment is set the button create can be clicked and a confirmation message will be appear. This ZIP file we can deployed as Web service on the services container using the dialog of deployment shown on the Figure 4.

Figure 4. jGASW service deployment tool



Tools re-location

The core functionality of jGASW beyond the description of CLI application is the instrumentation of the logic related with the interpretation of arguments, dependencies configuration and the execution. This operative process is organized in data management, the execution, and the results manipulation.

Data management

One fundamental feature of jGASW in case of file transfers is the delegation of this task to external tools. It means, the instantiated service does not transfer the results declared as files, and only provides references to them. The design of this feature responds to the necessity of avoiding the potential bottleneck of large amount of data transfers that certainly it is not part of the service execution. By contrast, if the service receives references to files as arguments, these are fetched to the effective point of execution managing the different protocols supported by jGASW, namely: file, http, https, ftp, lfn or gsiftp. After the execution of an application two scenarios are figured out regarding data. In the first scenario, if the resulting files are potentially inaccessible to the remote client jGASW puts available into a public space all the resulting files. Usually this context happens in a local execution where the outputs are defined using the file protocol so, a translation of the reference is possible in favor of other protocol such as http. In the second scenario, jGASW registers all resulting files product of a grid execution on a Storage Element, then references to those files are reported to the client. Naturally, in both cases files can be delivered to the client using an additional data transfer operation.

Execution

The instrumentation of the local execution interprets the description of the application building the command-line to execute. On the server, a isolated sandbox is created and the execution is then performed retrieving all necessary data to the sandbox and configuring the dependencies properly. The local execution is the simplest jGASW runtime instrumentation. The application runs in the same place where the service is hosted, for this reason multiple instances of heavy-demanding applications are not suitable and a remote relocation on a production grid is contemplated.

The grid execution allows several remote executions of an application transparently but requires remote relocation of resources. This execution implies use of several components of the grid infrastructure such as Workload Management System, Storage Elements, the Logging and Bookkeeping service, etc. The correct execution on the grid involves strategies from submission to monitoring procedure.

The final goal of jGASW is to provide an automatic execution type. Users without technical skills could find difficult the selection of execution type because they do not have a clear idea about the implications of relocation. Moreover users do not know the load status of the server nor the health of the grid. These arguments draw the necessity to provide an operation to choose automatically the type of execution in behalf of the user. The selection between local or grid execution is based on the application description, specific requirements and resource availability. The execution time and the requirement needs are the most important factors however, overhead or remote responses can also determine the type. The explicit operations still remain pertinents though.

Results manipulation

A service has arguments described with different data types and structures. In the same way the results of an execution should match the description of provided outputs. jGASW takes the original results of an execution and forwards them preserving that structure and data typing. The notions about such types and structures are explained in detail in the GWENDIA language proposal. Briefly, the input and output manipulation of data is based on the array programming principles. jGASW uses primitive types to map results with data types. Scalars and arrays represent all application parameters and results. Scalars are simple elements representing a plain primitive type. In contrast, (nested) arrays are groups of elements of same type structured according to the dimension of the array. Most of the time, when the results are not files they are presented in the standard output as sequences of strings. After execution is imperative to interpret these outputs using the description of arguments. This task involves parse, cast and map the result in the right structure and finally provide this structure as service result. jGASW reproduces as much as possible the structure organization resulting from an execution. Nevertheless this is not trivial and some agreement of structure is defined using delimiters of arrays and element separators. If an application provides a different result format the output should be adapted to a jGASW-compatible interpretation.

Invocation client interface

A generic client API to invoke services is the third element of the jGASW toolbox after the client used to wrap the CLI applications and the core libraries installed on the server to enable the instrumentation. The methods of the API parses the service description, interprets the contents and creates dynamically the messages to submit the information to the server. Besides, it is possible to use the same API to invoke external Web services interpreting the operations and arguments declared in the WSDL as long as such services met the requirements of data management used by the GWENDIA language. This capability of generalization let's workflow managers like Moteur use the API to build pipelines with jGASW services combined with other pre-existent services provided by third-parties.

Two parsers are implemented in order to obtain the list of services, ports and operations, the description of messages and sequences. A WSDL parser to handle the high level details of the service description, and a schema parser to manage the intrinsic details of primitive types as namespaces or dimensions of variables. From the final developers point of view only the fist parser is useful to obtain the sequence element of the operation to call. The schema parser is used by the WSDL parser internally. In the same way, the content details of the expected results are processed using the WSDL. In other words, a client can obtain all the necessary information to invoke and process an operation based on the description.

Using Web services the interoperability is granted between clients an server thanks to the messaging protocol independence. Consumers dispatch a well-defined message and wait for the result. This action is possible creating SOAP messages with the references retrieved from the description associated to their corresponding values and send them to the server. The jGASW API client implements a dynamic method to consume Web services because a generic procedure for invocation is mandatory. Each service is described using a specific and different WSDL therefore a direct manipulation of messages is done during execution. Similarly the process mechanism of the server response is performed, despite the different message formats since each server provides a message that is not necessary defined in the same manner. Nevertheless to pay special attention to that concern has limits because in practice is not possible to test all types of server messages. The jGASW approach works based mainly on the message responses of the reference implementation provided by Metro.

Use-case

Some advanced characteristics of jGASW cannot be observed at a glance. The manipulation of special file formats, hidden arguments or dependencies, and the interpretation of the resulting outputs requires an example to look in detail the necessity of the proposed description of resources and easy invocation.

BrainVISA is a software that allows users to trigger sequences of treatments in series of images. These treatments are performed by calls to command-lines. These tools, are the building blocks on which is built an assembly line. One of these applications for calculation of images is AimsLinearComb. It performs a sum of two brain activation maps. For instance, AimsLinearComb performs a linear combination obtaining a fusion of two binary functional-analysis activation in form of a new volume. An example of this command-line is:

Example 1. AimsLinearComb command-line example

```
$AimsLinearComb -i lwlebge.img -a 200.0
                -b 1.0 -j lwdupje.img
                -c 20.0 -d 1.0 -e 0.0
                -o lwtest.img
```

In practice this tool is more complex due to some suppositions the user should know:

- The input and output use Analyze images. This kind of image consists in two files with img and hdr extensions. In the command-line however, only the img file name appears explicitly representing the image.

- The tool execution produces a text file with the extension minf. This file is not expressed in the command-line but have to be included in the results with the Analyze image.
- AimsLinearComb needs several libraries for standalone execution. The user should configure the environment to include them in the list of the system. Typically this is possible in unix-like systems adding to the LD_LIBRARY_PATH the directory path where those dependencies are located.
- Additionally the user could be aware of the standard output or error messages, so all these concerns have to be considered when the tool is wrapped as service.

Using the jGASW interface, the creation and execution of this kind of CLI application as service is possible without adding intermediate scripts to adapt arguments manipulation or file formats interpretations. Although special attention to the description declaration is expected. The procedure is summarized as follows:

- Respect the order of the arguments, jGASW replace them in the same order. And take into account the case of letters for the options, extensions, and values because of case sensitiveness.
- Declare arguments 'a', 'b', 'c', 'd' and 'e' as Input of type double.
- Declare arguments 'i' and 'j' as Inputs of type URI setting the File Type value on the Advanced pane to Replace; and include the 'hdr' extension to the extensions text field.
- Declare the argument 'o' as Output setting the File Type value to Replace and include the hdr extension to the extensions text field as well.
- Declare an implicit argument (from Advanced pane) as Output and set the Value as regular expression checking the box. Then include the expression '.*minf' on the value field to get the generated text file in the list of results.
- Create a ZIP file containing the binary and all its dependencies and load it as attachment.

Thereafter the service is generated and then it can be hot-deployed on the services container using the deployment dialog. Finally, this service is ready for execution by client invocation.

Using the API for invocation, the service WSDL is interpreted with the description parser operations. The following arguments are identified as inputs: i1 with type anyURI; num1 and den1 with type double; i2 with type anyURI; num2, den2 and cst with type double.

The expected result values of AimsLinearComb are two elements: the text minf file and the Analyze image. By default jGASW add the contents of standard output and standard errors as files to the results. In short, after successful invocation a list of five file references are present by the service consumer.

Conclusions

the jGASW framework provides a rich description of CLI applications and execution procedure generating a Web service that encapsulates all the associated resources to be instrumented using different interfaces. This approach enables compositions of scientific workflows using Moteur2 with strong type mapping and complex structures. This solution is a step forward the conciliation of CLI applications with modern Service Oriented Architectures providing a clean and simple set of tools to assist scientists that are not computer specialists to build, run, combine, and share their work.