jGASW: A Service-Oriented Framework Supporting High Throughput Computing and Non-functional Concerns

Javier Rojas Balderrama, Johan Montagnat, Diane Lingrand

jGASW: a service-oriented framework supporting HTC and non-functional concerns

Javier Rojas Balderrama, Johan Montagnat, Diane Lingrand
University of Nice-Sophia Antipolis / CNRS, I3S, Sophia Antipolis, France. Email: {javier,johan,lingrand}@i3s.unice.fr

***Abstract*—Although Service-Oriented principles have been widely adopted by High Throughput Computing infrastructure designers, the integration between SOA and HTC is made difficult by legacy. *j*GASW is a framework for wrapping legacy scientific applications as Web Services and integrating them into an intensive computing-aware SOA framework. It maps complex I/O data structures to command lines and enables dynamic allocation of computing resources; including execution on local hosts or on grid infrastructures; data transfer management and support of non-functional concerns.**

***Keywords*-Legacy code as services; HTC services; SOA**

I. Introduction

The growth of the Internet in terms of size, reliability and bandwidth enabled *High Throughput Computing* (HTC) [1], in particular through cross-institutional Grids [2] that exploit large amounts of regular distributed computing resources. Seminally implementing the *Global Computing* (GC) model that shields the users from the heterogeneity and complexity of the underlying distributed system, grid middleware adopted Service-Oriented principles almost unanimously over the last years (WS-RF standard [3]) to deliver a flexible and adaptive support compatible with modern application development methodologies. However, the integration of GC and SOA models is not yet completed. This paper outlines the gap existing between these computing models and proposes a practical solution to bridge them.

As illustrated in top-left of Figure 1, in a GC environment, clients connect to an intermediate brokering service that acts as an proxy, handling the requests on their behalf and caching results. This model is very efficient to control and balance the overall system workload and therefore addresses well the needs of HTC. The broker implements a scheduler and/or resource allocator that optimizes the usage of the system. Managed resources are allocated temporarily for each computation task and have minimal system requirements. The GC model is easy to deploy and efficiently manages legacy command-line applications. Batch systems are good implementation examples, manipulating legacy applications as binary packages, transporting them to the computing resources, and executing them through remote command line invocation.

Conversely in a traditional SOA framework (bottom-left of Figure 1), services embedding the business codes are pre-deployed over a set of resources and invoked directly by clients through a standardized protocol. Clients first query a registry to locate the target(s) business service(s), and possibly balance workload among them. This *Meta-Computing* (MC) model has been implemented in Service-Oriented HTC solutions, such as GridRPC compliant middleware [4], [5]. Inheriting from SOA principles, the MC model provides flexibility, interoperability, autonomy, and re-usability. However, it requires business codes to be instrumented with a service interface as well as pre-deployement procedures that can be complex and frequent in large scale applications. Furthermore, clients are directly exposed to the communication with various resources and therefore they need to integrate complex concerns related to performance, reliability, fault-tolerance, security management, etc.

This paper describes the *j*GASW command-line application wrapper which implements the convergence between GC and MC as illustrated on right of Figure 1. It enables complex typed I/O data structures mapping, dynamic allocation of resources from high performance infrastructures, transparent local and/or remote execution, data transfers management, and non-functional concerns integration.

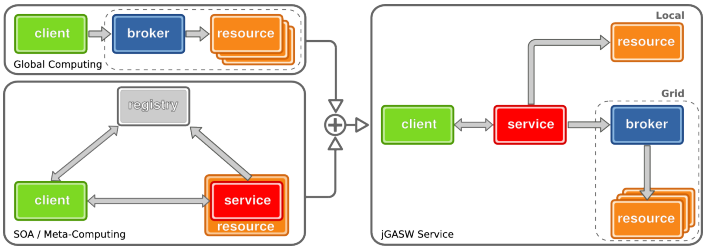

Figure 1. *j*GASW approach: convergence of Global Computing and SOA.

II. Requirements analysis and related work

Although non-functional core grid services (in charge of resources allocation, authentication, data transfers, etc) usually already benefit from a SOA, many scientific applications are developed as non-parallel, *Command-Line Interfaced* (CLI) business codes by domain experts. Due to the absence of an *Interface Description Language* (IDL), CLI are not well defined and often make assumptions on special file formats, implicit arguments, dependencies, and I/Os structure. To expose CLI applications as relocatable, self-consistent, remote invokable grid-aware services [6], an embedding process is needed that maps service message sequences, properly defined through an IDL, to the command lines. The resulting services are intended to be reused in

compute-intensive data analysis procedures described as *flows* of services. Grid workflow engines control concurrent services invocation exploiting the potential data and service parallelisms described through the workflow representation to deliver HTC.

Research initiatives aiming at integrating legacy business codes into SOA are usually targeting either domain-driven environments that provide ready-to-use frameworks or generic SOA that deliver interoperability and remote invocation through WS interfaces. Soaplab2 [9] provides the ACD metadata language to describe wrapped WS but it has no support for grid execution. The LONI Pipeline [7] and GEMLCA [8] are grid-aware but oriented towards building workflows in non-SOA environments. Opal2 [10] is a toolkit which enables the wrapping of CLI applications as WS on Grids and Clouds. It provides data management, and optional security enforcement. It proposes a global SOA solution, although it is domain specific in terms of data typing. GASW [12] and gRAVI [11] are respectively WS and WS-RF compliant wrapping tools but they both lack support of complex data structures representation. gRAVI is independent from a specific infrastructure and proposes both graphical and API for services invocation. gRAVI and Opal2 are dedicated to grid infrastructures and they do not enable local execution. Although demonstrating the need to bridge intensive computing models and SOA principles, most of these approaches require advanced technical skills and none cover the complete cycle including wrapping, deployment and execution of CLI.

In the rest of this paper, a real-world use case from the medical imaging domain (GWENDIA project, http://gwendia.polytech.unice.fr) is used to illustrate complex multi-dimensional structures description of CLI applications and a simple invocation mechanism. The application aims at estimating the myocardium deformation from time sequence of cardiac magnetic resonance image using non-rigid registration between consecutive image pairs [13]. The application workflow is composed of multiple codes dedicated to image intensity correction, region of interest identification, contours extraction, non-rigid registration, etc, and produces a dense motion vectors field for each time instant of the sequence. It features a compute intensive algorithm, too prohibitive for enactment on local resources. The workflow composition and enactment is handled by the MOTEUR engine, which is grounded on *array programming* principles [14] to handle complex data flows. The workflow description language of MOTEUR includes *iteration strategies* (defining how to iterate services invocation over the data segments) and *array nesting* (to dynamically assemble or disassemble data segments as requested by the application semantics and performance concerns). The CLI wrapper interface has to be highly expressive and flexible to cover all data flows composition capabilities.

Detailed description of each application parameter is vital for complete description of the workflows. For each embedded code, our framework generates a structured descriptor formalizing the application interface. The descriptor produced for the motion estimation code is partially represented in Figure 2. It is complex, due to different description needs.

```
<res:bundle res:category="application" res:osname="linux">
  <res:target>cme.sh</res:target>
  <res:version>1.0.0</res:version>
  <res:symbolicName>cme</res:symbolicName>
  <res:organization>CREATIS-LRMN</organization>
  <res:copyright>CNRS</res:copyright>
  <res:arguments>
    <res:argument res:io="in" res:type="URI"
          res:mapper="filesystem" res:implicit="true">
      <res:label>pyramids</res:label>
      <res:option/>
      <res:value res:regex="false" res:brand="replace"
        <res:content/>
        <res:extensions/>hdr</res:extensions>
      </res:value>
      <res:space>false</res:space>
      <res:nesting res:separator="," res:initDelimiter="␣"
          res:endDelimiter="␣">
        <res:dimension>1</res:dimension>
      </res:nesting>
    </res:argument>
    ...
    <res:argument res:io="out" res:type="URI"
          res:mapper="filesystem" res:implicit="true">
      <res:label>models</res:label>
      <res:option/>
      <res:value res:regex="true" res:brand="regular">
        <res:content>.*\.txt</res:content>
        <res:extensions/>
      </res:value>
      <res:space>false</res:space>
      <res:nesting res:separator="," res:initDelimiter="␣"
          res:endDelimiter="␣">
        <res:dimension>1</res:dimension>
      </res:nesting>
    </res:argument>
  </res:arguments>
  <res:dependencies>
    <res:dependency res:category="application" ...>
      <res:target>Motion_Estimation</res:target>
      <res:version>1.0.0</res:version>
      <res:symbolicName>cme_bin</res:symbolicName>
      <res:organization/>
      <res:copyright>CNRS</res:copyright>
    </res:dependency>
  </res:dependencies>
</res:bundle>
```

Figure 2. CLI application description using *j*GASW XML schema

First, inputs are list of Analyze-format images. This type of image volume is stored on disk into two files: image body (`img`) and header (`hdr` extension). The reference to the body file name is provided explicitly at run-time as an argument but the header is implicit. It is explicited in the descriptor through the `extensions` tags and the `brand` replace operation. Second, the execution produces fixed-name output text files that are not explicitly appearing on the command-line (attribute `implicit` is `true`). In the example, a regular expression (`regex`) is used to identify the relevant output file names to map (see the `value` of the second `argument`). The output is a one-dimension array of files (`dimension` is `1` in the `nesting` section). Third, the main processing script (`cme.sh`) depends on another binary (`Motion_Estimation`) as specified in the collection of `dependencies`.

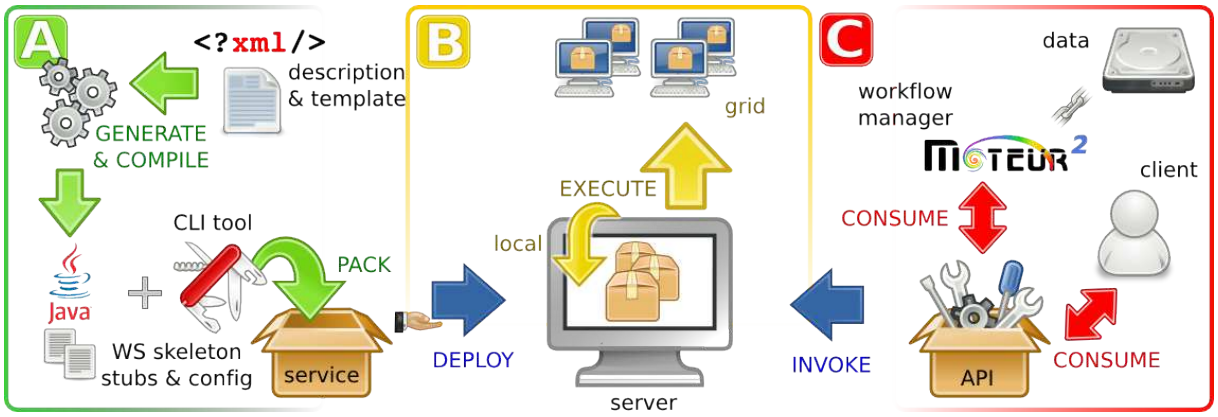

Figure 3. *j*GASW toolkit: (1) generation of service packages, (2) deployment and (3) invocation.

III. High performance SOA

Our method to integrate compute intensive application codes in a SOA framework is a java-based *Generic Application Service Wrapper* toolkit (*j*GASW) that provides a wrapper shell for CLI and a standard WS interface. *j*GASW is composed by three main parts, illustrated in Figure 3: (A) a service wrapper describing parameters of a command-line as service arguments; (B) a set of libraries to manage the allocation of resources and execution instrumentation; and (C) a programmatic interface (API) to invoke the wrapped applications. The wrapper and the libraries are based on a common data model. The API is an independent service consumer used on the client-side.

A. Description of CLI applications

The description of a CLI is based on a detailed XML schema covering technical information such as application name and operative system; declarative information to permit traceability such as version, symbolic name, application license; lists of arguments; and dependencies such as required external binaries and libraries. The parameters description need to be detailed enough to assemble the command line and manipulate I/O files. In *j*GASW the transformation of the description into a WS interface is based on a template engine that is used to generate the code and the configuration files needed to execute the CLI application through the service container. The generated code is compiled at run-time. It complies to JAX-WS, a WS implementation that makes extensive use of the java annotations mechanism. The generated code preserves the structures and data types defined in the description and represents a personalized skeleton of the WS and the file stubs used to serialize data between transactions. The application, its XML description, all its dependencies, compiled code and server-side configuration files are packaged into a Web application archive. To simplify the description file generation process and the service creation, *j*GASW includes both a command-line and a graphical interface.

*j*GASW stubs and skeletons map the service I/O arguments, preserving their data types and structure. The *j*GASW XML schema represents primitive data with four simple data types that are sufficient for describing CLI interfaces: `string`, `double`, `integer` and `URI` (references to data files). To adapt to the most complex data flows encountered and ease the declaration of data-parallel constructs, I/Os can be described as elements of homogeneous arrays, nested at any depth in case of multi-dimensional structures. In Figure 2 for instance, an input argument of type `URI` labeled 'models' is described. It is an implicit parameter, not associated to any option of the CLI but identified through a regular expression filter. It is treated as an ordered collection of files with a given dimension declared in the nesting tag. The computation results may be `URI`s associated to output data files and/or values printed in the process standard output stream. These outputs are interpreted through a non-trivial process taking into account their structure as documented in the application description (*e.g.* array delimiters and element separators). *j*GASW includes several mappers to parse, cast and map outputs to the appropriate data structures: a string mapper to interpret results printed on standard output; a file system mapper for the manipulation of set of files and directories produced; and a configurable mapper to manipulate application-specific structures.

B. Resources allocation

*j*GASW packages are instrumented to implement the server-side execution logic: interpretation of all arguments, dependencies configuration, data transfers and execution. This process adapts both to local and grid execution. It is extensible to support additional non-functional concerns.

Support for local and grid execution modes: The local instrumentation runs the application in the place where the service is hosted. This is useful for short execution

tasks which do not need intensive computing and would be penalized by the overhead introduced with grid submission. An isolated sandbox is created on the server to retrieve all necessary data and properly configure dependencies. The grid instrumentation performs remote executions of an application transparently allocating computing resources on the Grid through its Workload Management System (the broker in right of Figure 1). The WS wrapper submits computing tasks to the Grid and monitors their execution progress until completion before returning outputs to the client. Currently, *j*GASW supports execution on EGEE, the leading European Grid Infrastructure (http://www.eu-egee.org). Nevertheless, its design abstracts the notion of executor, to ease other middleware integration.

Data management: Files manipulated during execution need to be transferred to/from computing resources. For performance reasons, a dedicated file transfer protocol is preferred to transferring file content as part of service invocation messages. Furthermore, on the Grid, files are directly transferred between nodes and never transit through the workflow engine which would become a potential bottleneck in data-intensive applications. Only references to files are exchanged between the execution client and the *j*GASW service. Two scenarios are considered for output data files retrieval. For local execution, *j*GASW publishes files into a public space of the server host. A translation of the output file reference (*e.g.* server file path) is possible in favor of other protocols (*e.g.* public URL). For grid execution, *j*GASW registers all output files to a grid storage resource. In both cases files are delivered to the client using additional data transfer operations.

Integration of non-functional concerns: Additional non-functional concerns can be integrated within the *j*GASW wrapper shell to address specific deployment considerations without impacting the data modeling nor the core application. They are inserted as template macros and merged with the description during the generation of source code. *j*GASW natively includes non-functional concerns related to the Grid. The overhead introduced by grid submission and low reliability of grid infrastructures is impacting performance [16]. These issues may be overcome using adapted job submission strategies [17] such as (i) simple resubmission, time-outing and resubmitting abnormally delayed jobs; (ii) multiple submission, using a collection of the same job and canceling all but the first job that starts the execution to lower latency; and (iii) delayed resubmission, periodically submitting a copy of the job without canceling the previous submissions until at least one job starts executing. Fault tolerance is another concern since executions on the Grid face major inconveniences such as system incompatibilities (brokering mismatches) and resource unavailability. A simple strategy of white and black lists is implemented to select most reliable and faster grid resources. The *j*GASW framework is easily extensible: for example, access control, logging and accounting have been implemented to handle neurological studies [15].

C. *Generic invocation* API

Clients such as MOTEUR interface with the legacy application tools through the *j*GASW client API. It provides parsing and interpretation the WS description, and dynamic creation of the messages. It is generic and supports third-party WS as long as they comply to the array programming data structures. In Figure 2 for example, all resulting files with extension `txt` are returned to the client as a one-dimension array. The *j*GASW wrapper shields clients both from details of the CLI tools invocation and from the grid invocation interface (including data handling and grid security credentials management).

IV. Conclusions and discussion

*j*GASW is a rich framework to expose and invoke CLI applications as WS, bridging intensive computing and SOA. It provides a simple set of tools to assist non-computer specialist scientists to build, run, combine, and share their work. It covers advanced data flow manipulation capabilities and adopts a standard WS interface compliant with many workflow engines (*e.g.* Taverna [18], Triana [19] or BPEL engines). The integration of grid-related concerns into the application wrapper itself enables intensive computations, even for workflows enacted through non grid-aware engines.

Acknowledgements

This work is funded by the French National Research Agency (contract number ANR-06-TLOG-024, NeuroLOG project http://neurolog.polytech.unice.fr). *j*GASW is an open source toolkit (http://modalis.polytech.unice.fr/jgasw).

References

[1] D. Thain, *Conc. & Comp.: P. & Exp.*, 17(2-4):323-356, 2005.
[2] I. Foster, *ACM CCCS*, San Francisco, USA, 1998.
[3] I. Foster, *IFIC Intl. Conf. NPC*, Beijing, China, 2005.
[4] E. Caron *Intl J. HPCA*, 20(3):335-352, 2006.
[5] H. Nakada, "A GridRPC Model", GGF, Tech. Rep. July 2005.
[6] C. Mateos, *Soft. - Pract. & Exp.*, 38:523-556, 2008.
[7] I. Dinov, *Frontiers in Neuroinfofmatics*, 3(22), 2009.
[8] T. Delaitre, *Jnl of Grid Computing*, 3(1-2):75-90, 2005.
[9] M. Senger, *Bioinfo. OS Conf.*, Toronto, Canada, 2008.
[10] S. Krishnan, *SERVICES*, Los Alamitos, USA, 2009.
[11] K. Chard, *Intl. Conf. WS*, Los Angeles, USA, 2009.
[12] T. Glatard, *Future Generation CS*, 24(7):720-730, 2008.
[13] K. Maheshwari, *HealthGrid*, Berlin, Germany, 2009.
[14] J. Montagnat, *WORKS*, Portland, USA, 2009.
[15] A. Gaignard, *HealthGrid*, Berlin, Germany, 2009.
[16] D. Lingrand, *Parallel Computing*, 35(10-11):493-511, 2009.
[17] D. Lingrand, *HPDC*, Munich, Germany, 2009.
[18] T. Oinn, *Bioinformatics*, 17(20):3045-3054, 2004.
[19] I. Taylor, *Conc. & Comp.: P. & Exp.*, 17(9):1197-1214, 2005.