Everest: A Cloud Platform for Computational Web Services

Oleg Sukhoroslov and Alexander Afanasiev
Institute for Information Transmission Problems, Russian Academy of Sciences, Bolshoy Karetny per. 19, Moscow, Russia

Keywords: Service-Oriented Computing, Computational Web Services, Web Service API, REST, Service Framework, Cloud Platform, Platform as a Service.

Abstract: The ability to effortlessly reuse and combine existing computational tools is an important factor influencing research productivity in many scientific domains. While the service-oriented approach proved to be essential in order to enable wide-scale sharing of applications, we argue that its full potential in scientific computing is still not realized. In this paper, we present Everest, a cloud platform that supports publication, sharing and reuse of scientific applications as web services. The underlying approach is based on a uniform representation of computational web services and its implementation using REST architectural style. In comparison with existing work, Everest has a number of novel features such as the use of PaaS model, flexible binding of services with externally provisioned computing resources and remotely accessible API.

1 INTRODUCTION

Modern scientific research is often associated with complex computations and use of high performance computing resources. In their research scientists actively use software applications that implement computational algorithms, methods and models.

The ability to reuse existing computational tools is one of important factors influencing research productivity. However, such software often requires specific expertise in order to install, configure and run it that is beyond the expertise of an ordinary researcher. This also applies to configuration and use of high performance computing resources to run the software. Finally, researchers increasingly need to combine multiple tools in order to solve a complex problem, which brings an important issue of application composition.

The aforementioned problems can be addressed by provision of scientific applications in the form of remotely accessible, interoperable services. The use of service-oriented approach can enable wide-scale sharing, publication and reuse of applications, as well as automation of scientific tasks and composition of applications into new services (Foster, 2005). While the underlying principles of this approach are well-known, it is still an open question how to implement it in scientific computing in order to realize its full potential.

So far, most efforts in this area were focused on the provision of remote access to scientific tools via convenient web user interfaces. Examples of such approach include grid portals (Kacsuk, 2011), science gateways (Miller et al., 2010) and scientific hubs (McLennan and Kennell, 2010). While being successful among unskilled users, such systems do not actually expose applications as web services or provide programming interfaces thus limiting opportunities for application reuse, composition and integration with external applications.

This approach is in stark contrast to Web 2.0 applications and cloud computing services that support programmatic access via web service based APIs (Programmable Web, 2013). The proliferation of Web APIs has spawned development of mashups (Yu et al., 2008) that combine data, presentation and functionality from multiple services. Web service composition tools, such as Yahoo! Pipes, provided convenient interfaces for building mashups and making them available to everyone as new services.

This aspect is largely ignored in existing web-based scientific environments. As a rule, such systems do not provide tools for application composition or support workflows only on the level of computational jobs. The notable exception is Galaxy platform (Afgan et al., 2011) that supports tool composition and sharing of produced workflows. At the same time, Galaxy also doesn't expose tools and workflows as services thus limiting their use outside the platform.

Sukhoroslov O. and Afanasiev A..
Everest: A Cloud Platform for Computational Web Services.
DOI: 10.5220/0004941404110416
In *Proceedings of the 4th International Conference on Cloud Computing and Services Science* (CLOSER-2014), pages 411-416
ISBN: 978-989-758-019-2

Other efforts were focused on building tools for transformation of scientific applications into web services (Delaitre et al., 2005; Krishnan et al., 2009). These tools don't provide user interfaces beyond basic web forms for service invocation and rely on existing solutions for web service composition. While more powerful and well-aligned with SOA principles, this approach requires more effort in order to build a convenient environment for scientists. Service developers need an infrastructure to host services. The environment should provide mechanisms for service discovery, invocation and composition taking into account security requirements. These mechanisms should be accessible via convenient user interfaces facilitating the use of services for problem solving.

Both approaches also require considerable efforts to integrate an environment with high performance computing resources and grid infrastructures needed to run applications. As a rule, existing systems are tied to a single computing infrastructure and doesn't allow users to attach external resources.

The paper presents Everest, a cloud platform for computational web services that addresses discussed issues. It combines both approaches discussed above by exposing computational applications as web services with a uniform interface and implementing a web user interface for creating, sharing and accessing services. In contrast to previous work (Afanasiev et al., 2013), all functionality of the platform is provided remotely using the Platform as a Service model.

The paper is structured as follows. Section 2 discusses the model, interface and implementation of computational web services that underpin the proposed approach. Section 3 describes architecture and components of Everest platform in its current implementation. Section 4 concludes and discusses future work.

2 COMPUTATIONAL WEB SERVICES

2.1 Service Model

Computational web services (CWS) are the main entities managed by Everest. On the conceptual level, CWS represent a special type of web services targeted at processing computationally intensive requests. Such services should support management of long-running jobs and transfer of job data.

In contrast to generic web service interfaces to computing infrastructures, such as grids, CWS are specialized in running specific applications, i.e., solving specific classes of problems. Therefore a request to CWS normally doesn't contain an executable, but instead represents a set of input parameters describing a problem to be solved. We will refer to such requests as service-level jobs or just jobs. The job results can be represented as a set of output parameters in the same fashion.

It is responsibility of a CWS implementation to translate service-level jobs to one or more compute jobs submitted to underlying computing infrastructure in order to obtain desired results. Therefore CWS implement more specialized and high level interfaces than computing infrastructures. This makes it possible to hide the complexity of running compute jobs from service users and to enable transparent use of resources from multiple infrastructures.

In contrast to stateful web services typically found in enterprise systems, CWS process each incoming job in isolation. The only state managed by CWS is the state of processed jobs, so all data needed for a job should be provided in a request. While such restriction leaves out interactive session-based applications, it aligns well with the majority of computational tools such as solvers. This restriction also contributes to scalability properties of CWS.

The above description of CWS is rather general and can be related to a large class of services found not only in the scientific computing domain. Nevertheless, it explains the motivation and reasons behind implementation of CWS in Everest.

2.2 Service Interface

In technical terms, CWS can be implemented using any web service technology or style. Such freedom and lack of standards for implementation of CWS led to a multitude of approaches introduced by different systems. In order to facilitate reuse and composition of CWS implemented by different parties it is crucial to unify service interfaces.

The described model of CWS makes it possible to introduce a uniform service interface consisting of four operations:

- Job submission (as a set of input parameters);
- Retrieval of job state and results (as a set of output parameters);
- Job cancellation;
- Retrieval of service description (including description of input and output parameters).

All services implementing this interface support the same set of operations but can accept and return different sets of parameters. The last operation enables introspection of service parameters in order to

facilitate construction of job submission requests and processing of job results.

The described uniform interface follows an approach used by the HTTP protocol (Fielding et al., 1999) which defines a standard set of methods to indicate the desired action to be performed on the web resource identified by URI. This approach and the underlying REST architectural style (Fielding, 2000) proved to be essential to make the Web successful. In contrast, SOAP-based web services (Curbera et al., 2002) encourage creation of specialized interfaces and operations which provides a greater flexibility but complicates service reuse.

2.3 Interface Implementation

Using the REST architectural style the described uniform interface can be implemented as follows (Afanasiev et al., 2013). CWS represents a RESTful web service (Richardson and Ruby, 2008) identified by a Service URL. A job managed by the service is identified by a Job URL.

The Service resource supports the following HTTP methods:

- GET, which returns service description;
- POST, which performs job submission and returns a Job URL.

The Job resource created during job submission supports the following methods:

- GET, which returns the job state and results (if any available);
- PUT, which enables changing of the job state (e.g., job cancellation);
- DELETE, which destroys the job resource and deletes its data.

An additional File resource can be introduced to identify files passed to or returned by a service via its parameters. In such case parameter value contains a file URL. This enables passing large amount of data, which is particularly important for scientific computing, via appropriate data transfer mechanisms, such as HTTP, FTP or GridFTP.

Consider resource representation formats and means of describing service parameters.

The most widely used data representation formats for web services are XML and JSON. Among these JSON has been chosen for the following reasons. First, JSON provides more compact and readable representation of data structures, while XML is focused on representation of arbitrary documents. Second, JSON supports native integration with JavaScript language facilitating creation of web user interfaces for CWS.

The description and validation of service parameters can be accomplished by means of JSON Schema (JSON Schema, 2013), a de facto standard for defining the structure of JSON data.

2.4 Service Implementation

Consider an implementation of computational web service. Just like its interface, the inner workings of CWS follow a common pattern. A service listens to incoming job requests over HTTP and performs the following steps for each request:

- Authenticate and authorize the client;
- Parse and validate input parameters from the request;
- Translate input parameters to a compute job specification (executable, arguments, input and output files, etc.);
- Submit the compute job to configured computing resource;
- Monitor compute job state and provide this information to the client;
- Retrieve compute job results upon job completion;
- Translate compute job results to output parameters;
- Pass output parameters to the client.

Most of these steps can be implemented in the same fashion for any service disregarding its application domain. The only application specific parts are the ones that deal with processing of input and generation of output parameters. This makes it possible to implement a software framework, which provides a generic service skeleton that can be configured with the application specific parts (Afanasiev et al., 2013).

3 EVEREST PLATFORM

Everest is a cloud platform for computational web services that is based on considerations presented in the previous section. It implements a development framework and a hosting environment for CWS that adhere to the described uniform interface.

In contrast to traditional service development tools, Everest follows the Platform as a Service cloud delivery model by providing all its functionality via remote interfaces. A single instance of the platform can be accessed by many users in order to create, run and share services with each other without the need to install additional software on users' computers.

Another distinct feature of Everest is the ability to connect services with external computing

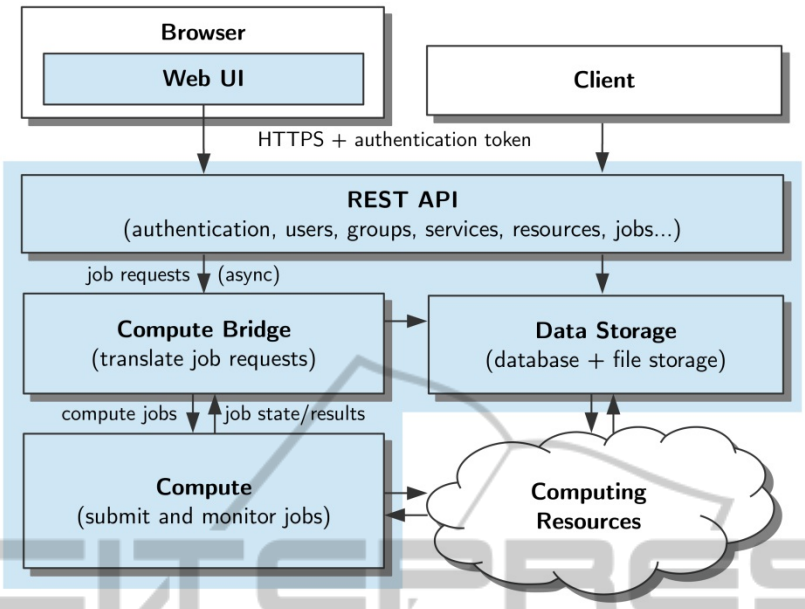

Figure 1: Architecture of Everest platform.

resources. That means that service developer can provide computing resource for running service jobs. This feature is useful in situations when platform's computing infrastructure has limited capacity or service developers need more control over an execution environment. A service user can also override the default resource by providing another resource for running her jobs.

The architecture of Everest is represented in Figure 1. Consider each of the platform's components in detail.

3.1 REST API

REST API is the platform's application programming interface implemented as a RESTful web service. It serves as a single entry point for all clients, including the web user interface.

The API includes operations for accessing and manipulating entities managed by the platform such as users, user groups, services, jobs and resources. In particular, the API implements operations from the uniform interface of CWS described in Section 2. It also provides additional operations related to services, such as service configuration and service discovery.

For each incoming request the API performs authentication of a client. The default authentication mechanism is implemented by means of OAuth bearer tokens (Jones and Hardt, 2012). A client can obtain a token by providing user credentials, i.e., username and password.

Upon successful authentication, the API also performs authorization of the requested action. Each entity managed by the platform has its' owner. The default security policy allows access to the entity only to its' owner. An owner can modify this policy, e.g., a service owner can specify a white list of users or user groups that are allowed to use the service.

The API relies on the data storage component to read and write information about platform entities. It also communicates with the compute bridge by passing it incoming job requests.

3.2 Web User Interface

Web user interface (Web UI) provides a convenient graphical interface for interaction with the platform. It is implemented as a JavaScript application that can run in any modern web browser without installation of additional software.

Web UI provides access to all functionality of the platform. It is built directly on top of the REST API, i.e., it uses the same interface as all other platform clients. While technically more challenging than traditional server-side web interface generation, this approach allowed us to reduce the server complexity and directly test the REST API.

The most important parts of Web UI are service configuration and job submission interfaces.

Service configuration interface is used to create new and edit existing services. It is implemented as a set of web forms that enable a user to specify all required information about a service including:

- Service metadata (name, description, etc.);
- Input parameters;
- Output parameters;
- Job template;
- Required files;
- Computing resource to run service jobs;
- Security configuration (white list, etc.).

Job template represents an application specific part of service configuration that is used by the platform to translate service requests to compute jobs. It includes the following information:

- Job command template that supports input parameter substitution;
- Mapping of input parameters to job input files;
- Mapping of job output files to output parameters.

Job submission interface is used to submit job requests to services. This interface is dynamically generated for each service according to the description of its input parameters. This information is also used to validate the request before its submission to the API. The implemented approach frees service developer from manual implementation of job submission forms.

3.3 Compute Bridge

Compute bridge is the core component of Everest that performs translation of service requests (service-level jobs) to compute jobs. It acts as a mediator between REST API and Compute subsystem that manages execution of compute jobs.

All job requests coming to REST API are asynchronously forwarded to the bridge. For each request the bridge performs translation of input parameters to a compute job specification according to the service configuration. The bridge also downloads input files that are referenced in the request.

A compute job specification produced by the bridge includes a command to be run, a list of job input files, a list of output files and a resource to run the job. The job specification is passed to the Compute subsystem for execution. The bridge also subscribes to notifications about the job state changes and translates these changes to the data storage.

Upon the job completion the bridge performs translation of job output files to output parameters according to the service configuration and saves the final result in the data storage.

3.4 Compute

Compute subsystem manages execution of compute jobs received from the bridge on computing resources attached to the platform. It performs all routine tasks related to staging of input files, submitting a job, monitoring a job state and downloading job results. All job state changes are translated to the bridge. Compute subsystem also monitors the state of all resources attached to the platform.

A computing resource can be attached to the platform by any user. A resource owner can configure a policy for accessing the resource. Any allowed user can bind the resource to any service.

Currently two approaches for integration with computing resources have been implemented. These approaches represent different tradeoffs between ease of integration and resource protection.

The first approach relies on existing remote access mechanisms supported by resources, such as SSH. In this case such mechanism is configured to accept credentials provided by the platform, e.g., SSH keypair. This enables the platform to directly execute any commands on the attached resource. Such approach makes it easy to attach computing servers or clusters without the need to install additional software on a resource. However, it also brings some issues. For example, sometimes it can be desirable to restrict commands that can be run by the platform, or a user can't provide full access to its account due to resource usage policy. This approach also doesn't support integration with resources that are not accessible remotely, such as desktop computers or resources behind a firewall.

The second approach addresses the mentioned issues by running a special agent on each attached resource. The agent acts as a mediator between the platform and the resource. This approach requires deployment of additional software on resources, but enables implementation of arbitrary security policies on the agent level and integration with resources behind a firewall. The communication between an agent and the platform is implemented through the WebSocket protocol (Fette and Melnikov, 2011). Upon startup an agent initiates connection with the platform to establish a bidirectional communication channel. This channel is used only for control and status messages. Job data transfer is performed by an agent via the HTTP protocol.

Currently the Compute subsystem doesn't perform resource selection during the submission of compute job. It is assumed that each service has only one resource linked with it. A user can override this resource with another one in a job request. In any case, the job specification passed to the Compute contains a single resource reference.

3.5 Data Storage

Data storage component implements long-term storage of information related to all entities managed by the platform. It is based on MongoDB (MongoDB, 2013), a document-oriented database system. Native support for JSON data structures with dynamic schemas proved to be useful during the platform development. The data storage also relies on GridFS feature of MongoDB for storing job data and other files.

4 CONCLUSIONS

The paper presented Everest, a cloud platform that supports development and hosting of computational web services. In comparison with existing work, Everest has a number of novel features such as the use of PaaS model, flexible binding of services with externally provisioned computing resources and remotely accessible API. While the platform doesn't provide its own infrastructure to run compute jobs as classic PaaS examples, it can handle the problems of resource allocation, job management, data transfer and so on without the interference of users.

Everest is work in progress. The platform is currently undergoing experimental evaluation and pilot deployment. The results of this work and application case studies will be presented in future publications. Future work will also address remaining gaps in platform's functionality and other challenges, such as development of programming APIs, supporting service composition, implementation of job scheduling mechanism enabling binding of multiple resources to a service, integration with grid infrastructures, and optimization of data transfer for services handling large amounts of data.

ACKNOWLEDGEMENTS

The work is supported by the Russian Foundation for Basic Research (grant No. 14-07-00309 A).

REFERENCES

Foster, I. (2005). Service-Oriented Science. *Science*, 308(5723), 814-817.

Kacsuk, P. (2011). P-GRADE portal family for grid infrastructures. *Concurrency and Computation: Practice and Experience*, 23(3), 235-245.

Miller, M. A., Pfeiffer, W., & Schwartz, T. (2010). Creating the CIPRES Science Gateway for inference of large phylogenetic trees. In *Gateway Computing Environments Workshop (GCE)*, 2010 (pp. 1-8). IEEE.

McLennan, M., Kennell, R. (2010). HUBzero: A Platform for Dissemination and Collaboration in Computational Science and Engineering. *Computing in Science and Engineering*, 12(2), pp. 48-52.

ProgrammableWeb (2013). *ProgrammableWeb - Mashups, APIs, and the Web as Platform.* http://www.programmableweb.com/.

Yu, J., Benatallah, B., Casati, F., & Daniel, F. (2008). Understanding mashup development. *Internet Computing*, IEEE, 12(5), 44-52.

Afgan, E., Goecks, J., Baker, D., Coraor, N., Nekrutenko, A., Taylor, J. (2011). Galaxy - a Gateway to Tools in e-Science. In: K. Yang, Ed. (ed) *Guide to e-Science: Next Generation Scientific Research and Discovery*, Springer, pp. 145-177.

Delaitre, T., Kiss, T., Goyeneche, A., Terstyanszky, G., Winter, S., Kacsuk, P. (2005). GEMLCA: Running Legacy Code Applications as Grid Services. *Journal of Grid Computing*, Vol. 3. No. 1-2, pp. 75-90.

Krishnan, S., Clementi, L., Ren, J., Papadopoulos, P., Li, W. (2009). Design and Evaluation of Opal2: A Toolkit for Scientific Software as a Service. In *2009 IEEE Congress on Services (SERVICES-1 2009)*, pp.709-716.

Afanasiev, A., Sukhoroslov, O., Voloshinov, V. (2013). MathCloud: Publication and Reuse of Scientific Applications as RESTful Web Services. In *Parallel Computing Technologies (PaCT 2013). Lecture Notes in Computer Science*, Vol. 7979, Springer, pp. 394-408.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). *Hypertext transfer protocol—HTTP/1.1*. Internet RFC 2616.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. dissertation, University of California, Irvine.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., & Weerawarana, S. (2002). Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *Internet Computing*, IEEE, 6(2), 86-93.

Richardson, L., & Ruby, S. (2008). *RESTful Web Services*, O'Reilly.

JSON Schema (2013). *JSON Schema and Hyper-Schema.* http://json-schema.org/.

Jones, M., & Hardt, D. (2012). *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC 6750.

Fette, I., & Melnikov, A. (2011) *The WebSocket Protocol*. RFC 6455, Internet Engineering Task Force.

MongoDB (2013). *MongoDB*. http://www.mongodb.org/