**КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2015 Т. 7 № 3 С. 593–599**

Ки&М

СЕКЦИОННЫЕ ДОКЛАДЫ

УДК: 004.75, 004.45

Development of distributed computing applications and services with Everest cloud platform

O. V. Sukhoroslov[a], A. O. Rubtsov, S. Yu. Volkov

Institute for Information Transmission Problems of the Russian Academy of Science, Kharkevich Institute, Bolshoy Karetny per. 19, build.1, Moscow, 127051, Russia

E-mail: [a] oleg.sukhoroslov@gmail.com

Received September 30, 2014

The use of service-oriented approach in scientific domains can increase research productivity by enabling sharing, publication and reuse of computing applications, as well as automation of scientific workflows. Everest is a cloud platform that enables researchers with minimal skills to publish and use scientific applications as services. In contrast to existing solutions, Everest executes applications on external resources attached by users, implements flexible binding of resources to applications and supports programmatic access to the platform's functionality. The paper presents current state of the platform, recent developments and remaining challenges.

Keywords: distributed computing, cloud platform, web services, REST, integration of computing resources, application composition

Создание распределенных вычислительных приложений и сервисов на базе облачной платформы Everest

О. В. Сухорослов, А. О. Рубцов, С. Ю. Волков

*Институт проблем передачи информации им. А. А. Харкевича Российской академии наук,
Россия, 127051, г. Москва, Большой Каретный переулок, д. 19, стр. 1*

Использование сервис-ориентированного подхода способно повысить производительность научных исследований за счет возможности публикации и совместного использования вычислительных приложений, а также автоматизации вычислительных процессов. Everest — облачная платформа, позволяющая исследователям с минимальной квалификацией публиковать и использовать научные приложения в виде сервисов. В отличие от существующих решений, Everest выполняет приложения на подключенных пользователями вычислительных ресурсах, реализует гибкое связывание ресурсов с приложениями и поддерживает программный доступ к функциональности платформы. В статье рассматриваются текущая реализация платформы, новые разработки и направления дальнейших исследований.

Ключевые слова: распределенные вычисления, облачная платформа, веб-сервисы, REST, интеграция вычислительных ресурсов, композиция приложений

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

Citation: *Computer Research and Modeling*, 2015, vol. 7, no. 3, pp. 593–599.

Introduction

The ability to effortlessly use and combine existing computational tools and computing resources is an important factor influencing research productivity in many scientific domains. However, scientific 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 computing resources to run the software. Finally, researchers increasingly need to combine multiple tools in order to solve complex problems, which brings an important issue of application composition.

While existing approaches and technologies provide solutions to some of these problems, they also have several drawbacks. Grid middleware uses service-oriented architecture to implement generic web services to access computing resources, however it is too low-level and hard to use for unskilled researchers. Scientific portals provide remote access to scientific applications and computing resources via convenient web interfaces, however they do not support programmatic access to applications thus limiting opportunities for application reuse and composition. Scientific web service toolkits support programmatic access by exposing applications as web services, however they lack common practices and require an infrastructure for hosting services.

Everest [Sukhoroslov, Afanasiev, 2014; Everest] is a cloud platform that addresses these problems by supporting publication, sharing and reuse of scientific applications across distributed computing resources. The underlying approach is based on combining the strengths of related approaches while eliminating the mentioned drawbacks by using modern web technologies and cloud computing models.

In contrast to traditional software, Everest follows the Platform as a Service (PaaS) cloud delivery model by providing all its functionality via remote web and programming interfaces. A single instance of the platform can be accessed by many users in order to create, run and share applications with each other without the need to install additional software on users' computers. Any application ported to Everest can be accessed via web user interface or unified REST API. The latter enables integration with external systems and composition of applications. Another distinct feature of Everest is the ability to run applications on arbitrary sets of external computing resources.

The paper presents current state of Everest, recent developments and remaining challenges. Section 2 provides an overview of Everest architecture and its main components. Section 3 discusses abstract model and implementation of Everest applications. Section 4 describes integration of external computing resources with Everest and binding of resources to applications. Section 5 presents Python API that enables Everest users to write programs that access applications and combine them in arbitrary workflows. Section 6 concludes and discusses future work.

Everest Architecture

A high-level architecture of Everest is presented in Figure 1. The server-side part of the platform is composed of three main layers: REST API, Applications layer and Compute layer. The client-side part includes web user interface (Web UI) and client libraries.

REST API is the platform's application programming interface implemented as a RESTful web service [Richardson, Ruby, 2007]. It includes operations for accessing and managing applications, jobs, resources and other platform entities. REST API serves as a single entry point for all clients, including Web UI and client libraries.

Applications layer corresponds to a hosting environment for applications created by users. Applications are the core entities in Everest that represent reusable computational units that follow a well-defined model described in the next section. Each application created by user is automatically exposed as a RESTful web service via the platform's REST API. This enables remote access to the application both via Web UI and client libraries. An application owner can manage the list of users that are allowed to run the application.

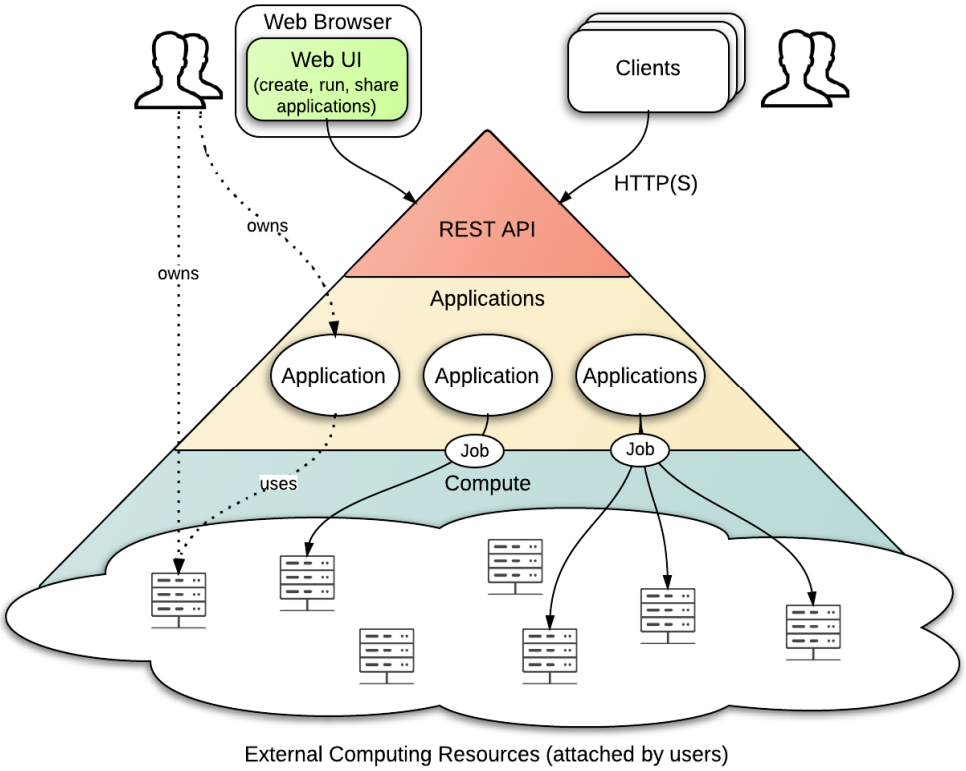

Fig. 1. High-level architecture of Everest

Everest doesn't provide its own computing infrastructure to run applications, nor does it provide access to some fixed external infrastructure like grid. Instead Everest enables users to attach to it any external computing resources and to run applications on arbitrary sets of these resources.

Compute layer manages execution of applications on remote computing resources. When an application is invoked via REST API it generates a job consisting of one or more computational tasks. Compute layer manages execution of these jobs on remote resources and performs all routine actions related to staging of task input files, submitting a task, monitoring a task state and downloading task results. Compute layer also monitors the state of resources attached to the platform and uses this information during job scheduling.

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. 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.

Client libraries simplify programmatic access to Everest via REST API and enable users to easily write programs that access applications and combine them in arbitrary *workflows*. At the moment, a client library for Python programming language is implemented.

Applications

Applications are the main entities in Everest - any computation is performed in the context of some application. Clients interact with applications by sending requests and receiving back results.

All Everest applications follow the same abstract model. An application has a number of *inputs* that constitute a valid request and a number of *outputs* that constitute a result of computation corresponding to some request. It is convenient to think of an application as a "black box" with some input and output ports or as a "function" with some arguments and return values. Just like pure functions, applications usually process each request independently from other requests in a stateless fashion.

The described model makes it possible to define a uniform web service interface for accessing applications [Afanasiev, Sukhoroslov, Voloshinov, 2013] which is essential in order to support application composition. This interface is implemented in Everest as a part of REST API.

From the user's viewpoint running an application basically means sending it a request containing input values and waiting for a result containing corresponding output values. For each request Everest performs the following actions:

1. Authenticate and authorize the client.
2. Parse and validate input values.
3. Create a new *job* which can be used to track the status of the request and to collect the result.
4. Translate input values to one or more *tasks* that represent units of computation.
5. Run job tasks on specified computing resources.
6. Translate job results to output values returned to the client.

Steps 1, 3 and 5 can be implemented in a similar fashion for all applications. However steps 2, 4 and 6 are application dependent. In order to simplify creation of applications, Everest provides generic implementations of these steps, the so called *application skeletons*, that can be configured by users. This declarative approach helps to avoid programming while adding applications to Everest.

In order to add an application to Everest a user should provide an application description that consists of two parts:

- *public information* that is used by clients in order to discover application and interact with it, including specification of inputs and outputs (also used to automatically implement Step 2 above).
- *internal configuration* that is used by Everest in order to process requests to the application and generate results (configuration of the application skeleton, application files and resource binding).

Currently Everest implements a single application skeleton for command-line applications which is suitable for porting to Everest existing applications. This skeleton generates jobs consisting of a single task. The internal configuration for this skeleton includes:

– a string template for mapping input values to a task command,
– input mappings that define how input values map to task input files,
– output mappings that define how task output files map to output values.

Additional skeletons for distributed applications with jobs consisting of multiple tasks, such as bag-of-tasks applications, composite applications (workflows), etc., are planned to be implemented in the future. The internal Compute layer of Everest already supports such jobs which is demonstrated by developing a generic application for running parameter sweep experiments [Volkov, Sukhoroslov].

Integration with Computing Resources

As was mentioned before, instead of providing it's own computing infrastructure Everest enables users to attach to it external computing resources and to run applications on arbitrary sets of these resources. From this point of view Everest can be seen as a multitenant metascheduling service.

Currently the preferred method for attaching a resource to Everest is based on using a developed program called *agent*. The agent runs on the resource and acts as a mediator between it and Everest. This method has one drawback - it requires deployment of the agent on each resource. However, it also brings a number of advantages in comparison to plain SSH access such as supporting resources behind a firewall and more strict security policies. Also the agent has minimal requirements (Python 2.6+) and is easy to install and run by an unprivileged user.

The agent supports integration with various types of resources via adapter mechanism. At the moment the following adapters are implemented:

– *local* — running tasks on a local server,
– *torque* — running tasks on a TORQUE cluster (agent is running on a submission host),
– *slurm* — running tasks on a SLURM cluster (agent is running on a submission host),
– *docker* — running tasks on a local server inside Docker containers.

The communication between an agent and the platform is implemented through the WebSocket protocol [Fette, Melnikov, 2011]. Upon startup an agent initiates connection with the platform to es-

tablish a bidirectional communication channel. This channel is used only for control and status messages. Authentication of an agent is performed by passing a secret token issued by Everest. Job data transfer is performed by an agent via the HTTP protocol.

In order to attach a resource to Everest a user should obtain a new resource token via Web UI, install the agent on the resource and run the agent with configuration including the obtained token. After the agent is connected to Everest it starts to send information about the resource state that is displayed in Web UI.

Besides standalone servers and clusters supported via the described agent, Everest also supports integration with the European Grid Infrastructure (EGI). This integration is implemented via EMI User Interface (UI). A user can attach as a new resource a specific EGI VO by providing a valid proxy certificate.

As with applications, a resource owner can manage the list of users that are allowed to use the resource to run applications. In order to run an Everest application it should be *bound* to at least one available resource. Everest implements flexible binding of resources to applications. Application owner can configure a static set of resources that should be used by Everest to run application tasks. In this case an application owner implicitly allows application users to run the application on these resources. Application owner can also enable *dynamic resource binding* when a user can manually select a resource for running his job. In both static and dynamic binding it is possible to specify multiple resources and let Everest to schedule application tasks across these resources.

Flexible resource binding opens new possibilities, but also brings some challenges. For example, if an application has commercial value or special hardware requirements, the application owner can restrict the application to run only on specific resources and disable dynamic binding. However, in such case the owner has to maintain resources used by application. On the other hand, if the application owner wants to share the application but doesn't have resources to support application users, he can enable dynamic binding and let users run the application on their resources. However, this brings another challenge of making the application portable across heterogeneous resources.

Programmatic Access

Running Everest applications via Web UI is easy and convenient, but it has some limitations. For example, if a user wants to run an application many times with different inputs, it is inconvenient to submit many jobs manually via web form. In other case, if a user wants to produce some result by using multiple applications, she has to manually copy data between several jobs. Finally, Web UI is not suitable if one wants to run an Everest application from his program or some other external application.

To support all these cases, from automation of repetitive tasks to application composition and integration, Everest implements a REST API. It can be used to access Everest applications from any programming language that can speak HTTP protocol and parse JSON format. However REST API is too low level for most of users, so it is convenient to have ready-to-use client libraries built on top of it. For this purpose a client library for Python programming language called Python API was implemented.

Figure 2 contains an example of program using Python API. It implements a simple diamond-shaped workflow (depicted in the top right corner of the picture) that consists of running four different applications — *A*, *B*, *C* and *D*.

At the beginning the program imports *everest* module which implements Python API and creates a new *session* by using a *client token*. Each client accessing Everest should present such token with its request in order to authenticate itself. In order to access applications the program creates a new *App* object for each application by passing application ID and the session.

The program initiates requests to applications by invoking *run()* method of the application object. Inputs are passed as a Python dictionary with keys and values corresponding to parameter names and values respectively. The *run()* method returns a *Job* object that can be used to check the job state and obtain the result. Note that *run()* method doesn't block the program until the job is done. Instead Py-

thon API performs all job related activities in the background thus allowing the program to continue its execution.

```
import everest

session = everest.Session(
  'https://everest.distcomp.org', token = '...'
)

appA = everest.App('52b1d2d13b...', session)
appB = everest.App('...', session)
appC = everest.App('...', session)
appD = everest.App('...', session)

jobA = appA.run({'a': '...'})
jobB = appB.run({'b': jobA.output('out1')})
jobC = appC.run({'c': jobA.output('out2')})
jobD = appD.run({'d1': jobB.output('out'), 'd2': jobC.output('out')})

print(jobD.result())

session.close()
```

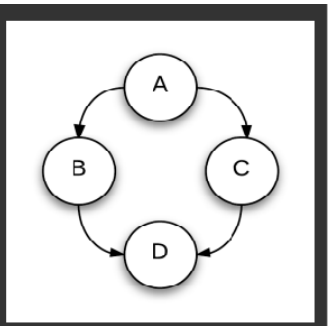

Fig. 2. Example program in Python API

In the presented example all jobs except *jobA* cannot be submitted immediately because they have data dependencies. Note how the program refers to the output values of the *jobA* by using *output()* method of the job and specifying the output names *out1* and *out2*. This method also doesn't block the program until the output value is available. Instead Python API will wait in the background until the *jobA* is completed, read the output values and then submit *jobB* and *jobC*. Similarly the *jobD* will be actually submitted to Everest only after *jobB* and *jobC* are completed.

The nonblocking semantics of Python API makes it simple to describe arbitrary workflows without requiring a user to implement boilerplate code dealing with waiting for and passing job results. This approach also implicitly supports parallel execution of independent jobs such as *jobB* and *jobC*.

After all jobs are created (while possibly not submitted) the program waits for a final result by calling the *result()* method on *jobD*. This method blocks the program until the job is completed and returns the job result. The result is returned as a Python dictionary with keys and values corresponding to output names and values respectively. Finally the program closes the session by invoking *close()* method. This terminates all background activities and ensures that the program exits normally.

Conclusions

The paper presented current state of Everest, a cloud platform supporting publication, execution and composition of applications running across distributed computing resources. The platform implementation provides all described functionality and is available online to all interested users [Everest]. Future work will address the remaining challenges such as implementation of advanced scheduling across multiple resources, publication of composite applications, supporting other types of applications (parallel, distributed, data-intensive) and integration with other types of computing resources (clouds, desktop grids).

References

Afanasiev A., Sukhoroslov O., Voloshinov V. MathCloud: Publication and Reuse of Scientific Applications as RESTful Web Services // Lecture Notes in Computer Science. — 2013. — Vol. 7979. — P. 394–408.

Everest. http://everest.distcomp.org/

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

Richardson L., Ruby S. RESTful Web Services. O'Reilly, 2007.

Sukhoroslov O., Afanasiev A. Everest: A Cloud Platform for Computational Web Services // 4th International Conference on Cloud Computing and Services Science (CLOSER 2014). — P. 411–416.

Volkov S., Sukhoroslov O. Running Parameter Sweep Applications on Everest Cloud Platform // In these Proceedings.