Providing Integration of UNICORE Services in Private PaaS Platform

Gleb Radchenko\textsuperscript{1} and Dmitry Savchenko\textsuperscript{1}

System Programming Department, South Ural State University, Chelyabinsk, Russia
E-mail: gleb.radchenko@susu.ru

The concept of grid computing has become a standard way of collaboration of scientific community computational resources while solving extra-large and resource-intensive tasks. On the other hand, the “cloud computing” concept is gaining increasing popularity in the field of provision of computing resources to an end-user on demand. PaaS (Platform as a Service) cloud model provides development, deployment, and administration tools to platform consumers, greatly simplifying the usage of remote computing resources. In this paper we propose the design and implementation of a Mjolnirr private cloud platform for development of the private PaaS cloud infrastructure. From a developer perspective, an application on the basis of the Mjolnirr platform is a set of independent components, which communicate through a message passing interface. Also, Mjolnirr platform allows integration of UNICORE services in cloud infrastructure, providing a mechanism to execute tasks in the UNICORE grid environment.

1 Introduction

In the last decade, the dominant method for providing remote computing resources to solve practical problems was the provision of such resources in accordance with the concept of cloud computing\textsuperscript{2}. From the point of view of developers and professional users, the models of IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) cloud solutions provide services that greatly simplify the usage of remote computing resources\textsuperscript{1}.

IaaS model is focused on providing consumers with low-level solutions, such as systems of processing, storage, and transmission of data, implemented in the framework of the concept of virtual machines. This approach assumes that each consumer of IaaS-resources will decide, what tools he will use to build and deploy his cloud application.

PaaS model provides a higher level of abstraction to the underlying computing resources, providing a transparent mechanism to deploy cloud applications using programming languages, libraries, services, and tools provided by the cloud platform. Thus, end users of PaaS solutions can significantly optimize the process of development, deployment, and execution of their applications in the cloud.

On the other hand, grid computing has become a standard way of collaboration of scientific community computational resources while solving extra-large and resource-intensive tasks. However, unlike electricity devices, integration of new resources to the grid-computing environment is not trivial.

To provide to the end user the simplicity of PaaS cloud solutions and computing power of existing grid systems, we propose a Mjolnirr cloud platform, providing the creation of private cloud PaaS-based systems, on the basis of component-oriented approach. Any library or Java application can be implemented on the basis Mjolnirr as a service. From a developer perspective, the Mjolnirr application is a set of independent services that communicate with each other via a message passing interface\textsuperscript{3}. Also, Mjolnirr platform allows
integration of UNICORE services in cloud infrastructure, providing mechanism to execute tasks in the UNICORE grid environment.

The paper is organized as follows: the next section presents an overview of the Mjolnirr platform architecture and implementation. Section 3 describes the process of integration of UNICORE services in cloud infrastructure and an example of application that uses UNICORE services via the Mjolnirr interface. Finally, in Section 4 possible future extensions and enhancements are discussed.

2 Mjolnirr Platform Architecture

The Mjolnirr platform provides infrastructure for cloud applications development, including software developer kit, message brokering system, and browser support. Java-based application or library can be implemented as a Mjolnirr-based service.

The Mjolnirr platform includes the following components (Fig. 1):

- **Proxy** provides access to the cloud system for the external clients and manages the communication between cloud application components. It also hosts all of the system services (built-in modules for user authentication, distributed file system, database access etc.). Proxy is the only component that is accessible from the external network.

- **Container** is responsible for hosting of cloud applications components and message transmission. It can be deployed both on personal computers and on the computing server nodes.

- **Components** are custom applications, developed to run in Mjolnirr cloud environment. Each component has a unique name. UI components (applications) are multi-page applications.

- **Clients.** All client applications use encrypted channel to communicate with the proxy. Each client should use a certificate for authentication.
Figure 2. Proxy architecture

2.1 Proxy

The Proxy component (Fig. 2) provides access to Mjolnirr system from the external network. The Proxy performs the following actions:

- it stores and provides static resources (page layout descriptions, images etc.) of the deployed cloud applications in the Static Storage;
- it acts as a messaging server for the components of cloud applications;
- it handles client’s requests to the Client Interface;
- it performs the authorization and authentication of users.

The external Proxy interface provides the following methods as RESTful API:

- `getUI(applicationName, pageName)` – returns the page layout description;
- `getResourceFile(applicationName, resourceName)` – gets the static file;
- `processRequest(componentName, methodName, args)` processes a client request, redirecting it to the first free suitable component. This method can be called directly from the application page (e.g. with JavaScript).

2.2 Container

The container (Fig. 3) provides cloud application component hosting. The container provides an API for remote components instances method invocation. The Mjolnirr installation can have any number of containers.

Any Mjolnirr-based application consists of independent components, which use a built-in messaging system, implemented in the basis of the Publisher-Subscriber pattern. The Proxy is responsible for message queue maintenance. The Message Server of the Proxy provides publisher-subscriber messaging service for cloud application components. Mjolnirr Containers subscribe to Message Channels that operate as a broadcast delivery. Any message sent to the Message Channel will be transmitted to the subscribers of this channel.

Each cloud application instance is subscribed on two types of Message Channels:
• **Component Public Channel:** every instance of the cloud application component is subscribed on this public channel. This a listener channel – when any message comes to this channel, the appropriate component instance will be invoked.

• **Instance Private Channel:** provides direct communication between instances.

When container starts, it performs several initialization steps. The order of the container initialization (Fig. 4):

- Container registers in the proxy database and receives the list of components to load in response;
- For each component from the list:
- The container checks its local cache, for each missing package container downloads it from the proxy;
- Container runs the component;
- Container subscribes on the Component Public Chanel and Instance Private Channel for the loaded component.

In addition, container has an opportunity to work in stand-alone mode. In this mode, the container does not support communication with other containers and acts as a stand-alone computing system (container and proxy at the same time).

2.3 Components

From the developer’s point of view, Mjolnirr cloud application is a collection of independent components communicating by message exchange. Components are represented as a package that contains the following information:

- *manifest*, that provides the interface of the component, including description of provided methods and their parameters;
- *executables* to handle incoming requests;
- *static files*, used in pages rendering (images, page layout descriptions, and scripts) for UI provision.

Each component can be:

- *Application Component*: provides the user interface definition, scripts, styles, and UI-specific actions. Optionally contains complex logic.

- *Module Component*: represents a single entity in the domain logic of the application. The Module Component provides data processing and storage, but does not provide interface and static files.

To implement Mjolnirr components API, we developed an interface definition language based on Java 6 annotation mechanism. Figure 5 shows an example of the interface for a Calculator component class.

A `@MjolnirrComponent` annotation denotes the interface class of component and contains the following fields:

- *ComponentName* - the name of the component in the system. Under this name the feature will be available for remote calls.
- *InstancesMinCount* - the minimum number of instances of a component in the system.
- *InstancesMaxCount* - the maximum number of instances of a component in the system.
- *MemoryVolume* - the amount of memory in megabytes required by the component for proper operation.
A \texttt{@MjolnirrMethod} annotation defines a separate method of the interface class as exportable. This annotation has one optional field: \texttt{executionTime}. The \texttt{executionTime} value indicates the maximum length of a method of execution of this method in seconds. The default value is 30 seconds.

Container class parses annotation interface for each downloadable component, reading the description of the methods provided by the component.

\section{Mjolnirr UNICORE Integration Module}

The Mjolnirr platform provides a UNICORE 7.0.2 integration module. UNICORE integration module is an interface for UNICORE installation (see Fig. 6).

UNICORE integration module has its own UNICORE certificate and works as an external client. The module does not use UNICORE authentication system, because Mjolnirr itself is not based on UNICORE, so the integration module is just a plug-in to execute tasks in the grid environment.

UNICORE component is an entry point to the whole UNICORE installation, in other words, it is the UNICORE Gateway. Integration module is based on UCC sources and uses standard UNICORE protocols to communicate with the grid environment. UNICORE integration module retrieves the list of all available sites and scans each site to find the proper application. When the application is found, UNICORE integration module uses the URL of the site, which hosts the target application (Fig. 7).

Mjolnirr UNICORE integration module provides the interface, shown in Fig. 8. As stated before, UNICORE integration module can be called in a standard Mjolnirr way. To send the task to the UNICORE installation, client application must submit the following attributes:

\begin{verbatim}
MjolnirrComponent{
    componentName = "calculator",
    instancesMinCount = 1,
    instancesMaxCount = 255,
    memoryVolume = 128
    \textbf{public class Calculator}
    \texttt{extends} AbstractApplication {
    \texttt{private} ComponentContext context;
    \textbf{@MjolnirrMethod}
    \texttt{public String calculate(}
        \texttt{String expression) }
        \texttt{return Helper.calculate(expression);} 
    \}
    \textbf{@Override}
    \texttt{public void initialize(}
        \texttt{ComponentContext context) } { 
    \texttt{this.context = context;}
    } 
}
\end{verbatim}

Figure 5. Component interface definition
appName - a name of the UNICORE application to run;

appVersion - a version of the application to run;

parameters - a map of the parameters to pass to UNICORE;

inputFiles - a list of files to send into the UNICORE installation (those files will be added to inputs automatically);

inputs - a list of files to be processed by the UNICORE-hosted application;

outputs - a list of files to be downloaded from UNICORE after application execution.

Those methods can be called in a standard Mjolnirr way, via built-in messaging system.
3.1 UNICORE Integration Test

To test the UNICORE integration module, we implemented an OpenMP test stand. Using this application, user can upload his own OpenMP program source to the platform and then get the detailed output for execution on N thread for N=1..16. An output example is shown in Fig. 9.

In this OpenMP test stand, the source file, which was uploaded by the user, is transmitted to the UNICORE application. This application does the following steps:

- Source file is compiled using GCC compiler with required flags (-fopenmp);
- Application sets the environment variables to specify current thread count;
- Application is run;
- Execution log is returned to Mjolnir.

4 Conclusions and Future Work

In this work, we presented a Mjolnir cloud PaaS platform, allowing development of scalable distributed Java-based applications on the basis of Publisher-Subscriber pattern message exchange. We developed a UNICORE integration module that allows using UNICORE services as components of such cloud applications. We presented the architecture, mechanisms and example of test Mjolnir application that use the UNICORE/X sites as cloudlets.
As a further development of the Mjolnir platform we will investigate and implement application-level migration support, integration with the advanced resource monitoring systems, flexible adaptation to load changes, advanced system security and application store. The application store will reduce the number of duplicate software products and simplify the creation of individual business infrastructure to meet the needs of a particular company.

Acknowledgements

The reported study was partially supported by RFBR, research project No. 14-07-00420-a and by Grant Fund for Assistance to Small Innovative Enterprises in Science and Technology research project No. 0000829

References