A Language-based Approach to Service Deployment

Xiaoning Wang[1, 2, 3] Wei Li[1, 4] Hong Liu[1, 2, 4] Zhiwei Xu[1, 4]
[1](Research Center for Grid and Service Computing, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100080, P.R. China)
[2](Graduate School of the Chinese Academy of Sciences, Beijing, 100039, P.R. China)
[3]wangxiaoning@software.ict.ac.cn [4]{liwei, liuh, zxu}@ict.ac.cn

Abstract

This paper presents a language-based approach to service deployment. The language is called Abacus, which is a service-oriented programming language for grid applications. In Abacus, a service is abstracted as a basic language construct, and service deployment is expressed by a deployment statement. This approach allows an Abacus application to automatically deploy a service at runtime without any configuration information. This paper compares the development detail in the Abacus environment with that in the manual deployment environment, and evaluates our deployment approach accordingly. On the one hand, this approach frees maintainers from annoying deployment work at low-level, which helps to enhance the productivity in building grid applications. On the other hand, this approach allows services to be deployed according to the application logic, helping to improve the utilization and the management of various grid resources.

Keywords: grid computing; service-oriented; service deployment; programming language; grid application

1. Introduction

As the evolution of Web Services and grid technology, services are becoming basic building blocks of grid applications. Service deployment is a critical step for developing or maintaining a grid application, because it determines the validity of the services to grid applications.

A solution to the service deployment problem needs to consider two factors: automation level and deployment cost. Though it is easy to deploy a service manually, an automatic approach could save some mechanical work, especially there are a large number of services to be deployed in an application. Deployment cost is the other factor, and it includes both human investment over the lifetime of the deployment and time and resource consumption when deploying.

Two papers [1, 2] introduced and compared four classes of approaches to service deployment, namely manual, script-, language-, and model-based deployment solution. They demonstrated a hypothesis that the level of automation pushes the cost up earlier in the development cycle to gain some benefits, and offered some guidelines for choosing proper solutions in the specific cases, subjecting to deployment needs.

This paper proposes a language-based approach to service deployment. Here, a service is defined as a piece of platform-independent software which provides some functions and can be deployed independently and addressed globally. The term "language-based" in this paper carries the following two meanings:

No configuration. Configuration information is usually needed for deploying a service manually or even by some tools. This suffers from two main drawbacks. Firstly, the service location needs to be determined by a maintainer or a developer. Although it is not included in the configuration information when deploying, it must be hard-coded in the application when invoking the service. Therefore, the programs may be modified frequently because of the changing requirements and volatile hosting environments. Secondly, there is no checking mechanism to guarantee the correctness of the configuration information. Some spelling errors in the configuration file may result in failures of the deployment, or even affect the container. For example, in Tomcat, once the element "className" in the deployment file is misspelt, the container will not start normally for it cannot load the specified classes. In this case, all the other services deployed in this container will not be available either. On the contrary, our language-based deployment approach requires no configuration, and the substrate system can deploy the specified service automatically. Thus, there is no need for either a maintainer or a

IEEE International Conference on Services Computing (SCC'06)
0-7695-2670-5/06 $20.00

developer to care the location of his services. Once compiled and debugged, a program can be executed in different hosting environments without modification. Moreover, the correctness can be guaranteed by the type-checking mechanism in the language. This approach frees developers and maintainers from much boring work on service configurations, especially when there are many services to be deployed in an application.

Deployment on demand. In complex grid applications, there may be many services involved, but which one will be actually accessed is up to the application logic at runtime. Therefore, deploying all the possible services beforehand will bring about unnecessary work and resource waste. However, our language-based service deployment approach allows a service to be deployed on demand at runtime. For instance, when the number of requests for a service exceeds some threshold, the service provider can dynamically deploy the same service on other nodes. Therefore, this approach is more flexible and helpful to improve the utilization of resources.

This deployment approach has been implemented in Abacus, which is a service-oriented programming language for grid applications. The introduction of the Abacus programming language appears in another paper [3]. Abacus abstracts a service as a basic language construct, and expresses the operations on services as its statements. Through abstraction, Abacus hides much low-level detail such as resource distribution and resource binding. Therefore, a different approach to service deployment is provided in Abacus, which is language-based and automatic, and expressed by a deployment statement. This deployment approach aims at freeing developers and maintainers from the troublesome work of service configuration and at improving the utilization of resources in the grid environment.

There is some related work on service deployment. The paper [4] also presents a language-based deployment approach. The language is named SmartFrog, which provides a framework for describing, deploying, igniting, and managing distributed applications. However, it focuses mainly on software package configuration and installation, not service deployment in the Web Services framework. Additionally, SmartFrog requires programmers to specify all the configuration information in their programs. As a result, that "language-based" carries a different meaning from ours. The paper [5] presents a hot deployment service, which meets one vital requirement for building an ad hoc grid environment in conformance with OGSA[6]. It mainly aims at deploying grid services onto a grid node without disrupting other service instances which are already running there. Another related work is GlassFish[7], which is an application server. GlassFish utilizes annotation [8, 9], a new feature of Java EE 5, to simplify the deployment and management of a service. In this framework, the configuration information is coded in the Java source files instead of in the configuration file, and can be recognized and handled automatically by the runtime system.

The rest of this paper is organized as follows. In the next section we introduce the language-based service deployment mechanism in Abacus. Section 3 describes a use case for experiments. Section 4 compares the development detail in the Abacus environment with that in the manual deployment environment, and our deployment approach is evaluated in Section 5. We summarize the paper in Section 6, and discuss the future work on the Abacus language.

2. Service Deployment Mechanism in Abacus

The service deployment mechanism in Abacus includes the following three key points. Firstly, a service is abstracted as a language construct, and it is a primitive operating unit of grid applications. Secondly, services in Abacus are organized in a virtual service space[3], where each service is identified by a location-independent virtual address; Thirdly, service deployment is expressed by a statement in the language without any configuration, and the substrate system can deploy a service at runtime according to the application logic.

2.1 Service Abstraction

Service abstraction is the basis of our deployment mechanism. The declaration syntax of a service in Abacus is the following BNF, which is similar to a class declaration in Java:

ServiceDeclaration:
  service *ServiceName ServiceBody*
  static service *ServiceName ServiceBody*
ServiceBody:
  { *ServiceBodyDeclarations* **}**
ServiceBodyDeclarations:
  ServiceBodyDeclaration ServiceBodyDeclarations
  ServiceBodyDeclaration
ServiceBodyDeclaration:
  FieldDeclaration
  ConstructorDeclaration
  MethodDeclaration
MethodDeclaration:
  PublishedMethodDeclaration
  PrivateMethodDeclaration

The keyword "static" is used to declare the deploy scope of a service. The service declaration with "static" keyword defines a stateful service, which maintains state information between message calls, and is used for encapsulating resources. While the service declaration without "static" keyword defines a stateless service that does not maintains state information, and requires less resource overhead.

2.2 Service Virtualization

Service virtualization provides a location-independent feature for grid applications. Services in Abacus are organized in the virtual service space. Each service is addressed by its virtual address, which is a 32-byte long Universally Unique Identifier (UUID)[10]. Therefore, when an Abacus program moves from one site to another, other programs can still access its services through the virtual addresses.

The virtual service space is managed and maintained by the Abacus compiler and the Abacus runtime system, thus service locations are transparent from either a maintainer or a developer.

When compiling, the compiler will assign a virtual address for each shared service declared in the source files, and record them in an interface file. When other customers want to use these shared services, they need to compile their programs with this interface file, in order that the system can get the virtual addresses from it. At runtime, the substrate system manages and maintains the mappings from the virtual addresses to the service physical addresses, and can automatically locate the services through their virtual addresses. Therefore, from the point of programmers, they do not have to care where their services are located.

2.3 Service Deployment

Service deployment in Abacus is expressed by a deployment statement:

NewExpression:
New *ServiceName* **( );**

This statement automatically deploys a service named *ServiceName* to a grid node, and returns a virtual address. After this statement is performed, the service just deployed can be accessed immediately by the same application or other applications. In this operation, most of the service configurations are automatically handled by the substrate system of Abacus without any additional information, including the location of the service, the transport method, the constructing of messages, etc.

An implementation of the substrate system of the Abacus programming language is Abacus virtual machine (AVM)[11], which is a language-level virtual machine for grids. Abacus virtual machine provides a set of service-oriented instructions to manage the life cycle of a service.

There are three key components in the AVM, namely execution engine, resource router, and service container. Execution engine takes charge of maintaining the AVM run-time data structures and processing all the instructions and native methods. Resource router maintains the mappings from the service virtual addresses to their physical addresses, and locates the accessed services. Service container mainly handles messages between services.

AVM provides a deployment instruction to deploy a service. In terms of runtime context of the application, AVM can decide where to deploy the service, how to handle the messages, etc. In this operation, AVM firstly allocates resources and assigns a physical address for the service. Then, it builds a mapping from its virtual address to the physical address in the resource router. After that, AVM returns the virtual address of the service to the program.

3. A Use Case

We take NGB (NAS Grid Benchmarks)[12] as our use case for the experiments. NGB is a benchmark suite for grid computing, which is proposed by NASA. It evolves from NPB (NAS Parallel Benchmarks)[13], which has been widely used in parallel computing systems.

3.1 Introduction to NGB

In NGB, there are five basic tasks (i.e. BT, FT, LU, MG, SP) on behalf of various types of scientific computation. These tasks interact with each other in different forms, and construct four problems representing four typical classes of grid applications. The problems are *Embarrassingly Distributed* (ED), *Helical Chain* (HC), *Visualization Pipe* (VP), and *Mixed Bag* (MB) respectively. ED requires no communication, and all tasks are executed independently. It tests basic functionality of grids and does not tax their communication performance. HC is totally sequential. Hence, any time spent on communicating data between two tasks is fully exposed. VP requires specifying concurrent execution of tasks with nontrivial dependencies. It allows pipelining and overlapping communication times with computational work. MB is similar to VP, but the tasks have different amounts of computational work.

The implementation routines of NGB are similar using any development tool or on any grid infrastructure. Different problems in the NGB are defined in the data flow graphs, whose nodes and directed arcs represent computation tasks and their communication channels respectively. Each task in the problem is encapsulated as a service. At the beginning, the client starts all the services in the problems. Each service waits and gathers the data from its incoming services for initializing, performs the specified task in the request, and then sends the results to its outgoing services. At last, the client gets all the results from services, verifies them, and generates a benchmark report.

Job turnaround time is one of the most important quantitative metric in this benchmark; however, it is not our concern. In the following of this paper, our focus is the development detail rather than the metric results of the benchmark.

3.2 NGB Features

We choose NGB as our use case because of its following features.

NGB's scale is large enough. The problems in NGB range over five classes in size, namely S, W, A, B, and C. For example, the problem in class B contains 18 tasks, each of which is encapsulated in a service. Likewise, the problem in class S contains 9 tasks. This scale is indeed not very large, but it is enough to show us the troubles brought about by the traditional deployment approach.

The communication scheme in NGB is complex enough. Different problems in NGB represent different types of grid applications. The interaction in these problems (ED, HC, VP, and MB) varies from easy to complex. For example, ED has no communication between services, while MB requires pipelining and overlapping communications between services.

The messages in NGB are complex enough. The messages exchanged in NGB contain many complex data structures. On the one hand, to handle these complex data structures, including large-size arrays, a large number of resources, especially memory, are required. Therefore, NGB can effectively test a system's ability to handle such messages. On the other hand, it is necessary to consider type mappings when forming and parsing these messages.

4. Experiments

4.1 Environments

We developed the NGB in two environments, and compared the detail. One was the Abacus environment. Abacus programs were performed on the AVM. The other was the traditional manual deployment environment, and we took Java (JDK, version 1.4.2_07) as our development language, with Axis (version 1.2RC) and Tomcat (version 4.1.30) as the application server.

Both environments were performed in a LAN testbed. The LAN testbed consists of six PC servers each with dual Intel Xeon 2.4GHz processors and 1GB RAM. The servers are connected by a 100 Mb Ethernet. The operating systems on these servers are Redhat Linux with kernel 2.4.18.

4.2. Comments on Data Structures

In the NGB implementation, there are some required data structures. Some data structures, including *DGraph*, *DGNode*, *DGArc*, are used to describe a graph which defines a problem in NGB. *BMRequest* implements request data, which is used to start up a task through a Benchmark Server. *BMResults* is used to report the results of an NGB task. *ArcHead* implements on server side a data source structure transmitting data along a DFG arc.

4.3. Development in Java

The dataflow graph required by NGB is recorded in a file, as illustrated in Fig. 1. In this file, a node represents a computation task, specified by the task's id, type, size, and location. An edge represents an interaction channel between the services, specifying its source and target nodes. This file is used to form the *DGraph* structure in the program.

It is important to develop a service which encapsulates a computation task in NGB. The service contains a couple of interfaces. One is used to start the whole computation task, invoked by the NGB client, and the other is used to get input data from other services. Therefore, a NGB service also plays the role of a client of other services.

As the role of a service, necessary configuration information is included in a service configuration file of the container, which is used for deployment (in Fig. 2).

There are some large-size arrays encoded in the message body. On the one hand, the messages will be too long after XML-based encoding, so it spends much time to transmit. On the other hand, much memory is

required to handle the messages accordingly. To solve this problem, these large arrays are transferred as the attachment of the SOAP message and encoded in the bytecode format. This is coped with by a handler specified in the configuration (Fig. 2). For type mapping, we adopt beanMapping, which provides a default serializer and deserializer. This requires that all the mapped classes here should follow the standard of Java Bean.

```
graph: {
title: "VP.S"
node:{title: "0" label: "BTTask.S.cngrid5.linux.ict.ac.cn:8085"}
node:{title: "1" label: "MGTask.S.cngrid5.linux.ict.ac.cn:8085"}
node:{title: "2" label: "FTTask.S.cngrid5.linux.ict.ac.cn:8085"}
node:{title: "3" label: "BTTask.S.cngrid3.linux.ict.ac.cn:8085"}
node:{title: "4" label: "MGTask.S.cngrid3.linux.ict.ac.cn:8085"}
node:{title: "5" label: "FTTask.S.cngrid3.linux.ict.ac.cn:8085"}
node:{title: "6" label: "BTTask.S.cngrid4.linux.ict.ac.cn:8085"}
node:{title: "7" label: "MGTask.S.cngrid4.linux.ict.ac.cn:8085"}
node:{title: "8" label: "FTTask.S.cngrid4.linux.ict.ac.cn:8085"}
edge:{sourcename: "0" targetname:"1"}
edge:{sourcename: "0" targetname:"3"}
edge:{sourcename: "1" targetname:"2"}
edge:{sourcename: "2" targetname:"5"}
edge:{sourcename: "3" targetname:"4"}
edge:{sourcename: "3" targetname:"6"}
edge:{sourcename: "4" targetname:"5"}
edge:{sourcename: "5" targetname:"8"}
edge:{sourcename: "6" targetname:"7"}
edge:{sourcename: "7" targetname:"8"}
}
```

Figure 1. Dataflow Graph in Java-based NGB

```
<service name="benchserver" provider="java:RPC" >
<requestFlow>
   <handler type="java:tasks.GetdataHandler"/>
</requestFlow>
  <parameter name="allowedMethods" value="*"/>
  <parameter name="className" value="tasks.BenchServer"/>
  <parameter name="scope" value="application"/>
  <beanMapping languageSpecificType="java:tasks.BMRequest"
        qname="ns:BMRequest" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.BMResults"
        qname="ns:BMResults" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.BMArgs"
        qname="ns:BMArgs" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.SparseA"
        qname="ns:SparseA" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.Datestub"
        qname="ns:Datestub" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.DGraph"
        qname="ns:DGraph" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.DGNode"
        qname="ns:DGNode" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.DGArc"
        qname="ns:DGArc" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.ArcHead"
        qname="ns:ArcHead" xmlns:ns="http://tasks"/>
 </service>
```

Figure 2. Service Configuration for the Service

As the role of a client, a configuration file is also required (in Fig. 3) to assist in forming and parsing the messages on the client side. When invoking a service, we use the class *FileProvider* to specify the client configuration file, and the service invocation codes are showed in Fig. 4.

After the service is coded, the compiled class files are archived into a jar file, which is required to be located at the specified location of the Tomcat.

The NGB client is relative simple. It starts each NGB server, gathers all the results, and prints a report. The service invocation codes are similar to that in Fig. 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
    name="defaultClientConfig">
<globalConfiguration>
  <requestFlow>
    <handler type="java:tasks.PutdataHandler"/>
  </requestFlow>
</globalConfiguration>
 <transport name="http"
     pivot="java:org.apache.axis.transport.http.HTTPSender"/>
 <transport name="local"
     pivot="java:org.apache.axis.transport.local.LocalSender"/>
 <transport name="java"
     pivot="java:org.apache.axis.transport.java.JavaSender"/>
  <beanMapping languageSpecificType="java:tasks.BMRequest"
     qname="ns:BMRequest" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.BMResults"
     qname="ns:BMResults" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.BMArgs"
     qname="ns:BMArgs" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.SparseA"
     qname="ns:SparseA" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.Datestub"
     qname="ns:Datestub" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.DGraph"
     qname="ns:DGraph" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.DGNode"
     qname="ns:DGNode" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.DGArc"
     qname="ns:DGArc" xmlns:ns="http://tasks"/>
  <beanMapping languageSpecificType="java:tasks.ArcHead"
     qname="ns:ArcHead" xmlns:ns="http://tasks"/>
</deployment>
```

Figure 3. Client Configuration for the Client

```
String endpoint = "http://"+lreq[i].MachineName
      +"/axis/services/benchserver";

Service service = new Service(new FileProvider("usr.wsdd"));
Call call = null;
call = (Call) service.createCall();

call.setProperty("SparseA",spacon);

call.setOperationName(new QName(endpoint, "SendData"));
call.setTargetEndpointAddress(new java.net.URL(endpoint));
call.invoke(new Object[]{lreq[i],res});
```

Figure 4. Service Invocation in Java

Before the client program runs, it should make sure that all the nodes specified in the dataflow graph file have started their service container (Tomcat in this use

case). When the program starts, all the tasks will be performed on the specified nodes. In the execution of each service, the runtime system is Java virtual machine, and the loading and management unit of the system is a class.

4.4. Development in Abacus

Like the case in the Java environment, a dataflow graph file is also required to define the NGB problem (an example in Fig. 5). The id, type, and size of each task and the two endpoints of each communication channel are specified in this file.

```
graph: {
title: "VP.S"
node:{title: "0" label: "BTTask.S"}
node:{title: "1" label: "MGTask.S"}
node:{title: "2" label: "FTTask.S"}
node:{title: "3" label: "BTTask.S"}
node:{title: "4" label: "MGTask.S"}
node:{title: "5" label: "FTTask.S"}
node:{title: "6" label: "BTTask.S"}
node:{title: "7" label: "MGTask.S"}
node:{title: "8" label: "FTTask.S"}
edge:{sourcename: "0" targetname:"1"}
edge:{sourcename: "0" targetname:"3"}
edge:{sourcename: "1" targetname:"2"}
edge:{sourcename: "2" targetname:"5"}
edge:{sourcename: "3" targetname:"4"}
edge:{sourcename: "3" targetname:"6"}
edge:{sourcename: "4" targetname:"5"}
edge:{sourcename: "5" targetname:"8"}
edge:{sourcename: "6" targetname:"7"}
edge:{sourcename: "7" targetname:"8"}
}
```

Figure 5. Dataflow Graph in Abacus-based NGB

For deploying a service, the Abacus language provides a deployment statement *new*. The statement in Fig. 6 deploys a service named *BenchServer* on some grid node. The syntax of invoking a service is similar to object invocation syntax (in Fig7).

```
server=new BenchServer();
```

Figure 6. Service Deployment in Abacus

```
nxtServ.SendData(lreq[i],res);
```

Figure 7. Service Invocation in Abacus

When the client program starts, all the service will be deployed on grid nodes automatically according to the application logic. In Abacus, the basic loading and management unit of the system is a service, and the runtime system decides when to collect garbage services.

4.5. Overhead

We ran the NGB on our testbed, and measured the time cost on our service deployment. To exclude the factor of the network, we configure the AVM to deploy all the services in the local node. The average time spent on service deployment in NGB is 98.22 μ s. Deploying a service which does nothing costs about 21 μ s. These values reflect the overhead of our approach, and they can be improved with the improvement of the substrate system of Abacus.

5. Evaluations

From the above experiments, we summarize the differences between developing in the Java environment and that in the Abacus environment in Table 1.

From Table 1, we can see that Abacus saves many codes and steps in developing and deploying grid applications. The reason lies in two aspects.

Firstly, Abacus is a programming language particularly designed for developing grid applications. Therefore, much work, such as type mapping and message constructing, can be automatically dealt with by the substrate system of Abacus.

Secondly, the language itself provides rich expressions for service-oriented applications. On the contrary, Java is designed to express objects and classes, so developers are confronted with annoying transformation when expressing services using Java objects.

One of the most important principles in Abacus is abstraction. It is abstraction that helps Abacus to hide much deployment information, and makes developers flexibly express services and service operations.

AVM also provides a framework to manage the lifecycle of a service. It can replace Java virtual machine and application server layer such as Tomcat to decrease the layers of the system and avoid version confliction of multiple layer tools.

Table 1. Differences Summary

	Java plus tools	Abacus
Knowledge of developers or maintainers	Java, XML, SOAP, WSDL, and WSDD, or maybe even more.	The Abacus language.

Location transparency	No. It is required to specify the physical address (URL) of the service.	Yes. There is no need to specify which node tasks are performed.
Lines of service configuration	In the above case, there are 26 lines of configuration for the service and 34 lines for the client.	There is no service configuration file.
Primitive construct in a program	All the service codes are expressed in the form of Java classes, including *Call*, *Service*, *FileProvider*, etc.	A service is a primitive construct when writing codes.
Deployment time	All the services must be deployed before the application is performed.	It is allowed to deploy a service in terms of application logic when the application is running.
Lines of one service invocation	At least 8 lines of codes.	Only one line of codes.
Management entities in the running system	Classes and objects.	Services.

The advantages of this language-based service deployment approach can be summarized as follows:

Easy to learn. Beginners do not have to grasp much knowledge such as XML, WSDL, or WSDD, and what they only need to learn is Abacus, a language similar to Java.

Location-independent access. Our approach hides service location information, which allows users to access a service in a location-independent style. When a service moves to a different node, its client code doesn't have to be modified.

No service configuration. Maintainers do not have to do much low-level annoying work. Especially when configuring services, they are not required to provide any service configuration information. This feature reduces such risk that spelling errors in configuration information may lead to troublesome exceptions when debugging an application.

On-demand deployment. This language-based approach to service deployment allows developers to deploy a service on demand, that is, the deployment of a service may be determined according to the runtime states or some other conditions of the application. It makes the language more suitable for complex applications.

Supports for service management. Service is a key element in Abacus, and AVM manages the lifecycle of a service, including collecting garbage services. Therefore, this language-based deployment approach helps to effectively utilize and manage various grid resources.

6. Conclusions

In this paper, we sketched a language-based approach to service deployment in the Abacus programming language. We compared the development in the Abacus environment with that in the manual deployment environment, and evaluated this deployment approach accordingly. With abstraction provided at language-level, Abacus provides a more flexible approach to deploy a service. In this approach, there is no need for a developer to care any low-level configuration of a service, and it is allowed to deploy a service on demand at runtime. Therefore, this approach helps to decrease the cost of building a grid application, and to make better use of various grid resources.

To make Abacus complete and more useful, much is to be done in the future. For example, exception handling will be considered in the language, for it can encapsulate low-level failure and report a readable exception message. We will also provide a set of effective libraries to assistant programming.

Acknowledgement

This work is supported in part by the National Natural Science Foundation of China (Grant No. 60573102, 90412010), the 973 Program (Grant No. 2003CB317000, 2005CB321800), and the Chinese Academy of Sciences Hundred-Talents Project (Grant No. 20044040). The authors are very grateful for these supports.

References

[1] V. Talwar, D. Milojicic, Q. Wu, C. Pu, W. Yan, and G. Jung, "Approaches for Service Deployment," IEEE Internet Computing, vol. 09, no. 2, pp. 70-80, 2005.

[2] V. Talwar, Q. Wu, C. Pu, W. Yan, G. Jung, and D. Milojicic, "Comparison of Approaches to Service Deployment," presented at Proceedings of the 25th IEEE Internet Conference on Distributed Computing Systems(ICDCS'05), Columbus, Ohio, USA, 2005.

[3] X. Wang, L. Xiao, W. Li, H. Yu, and Z. Xu, "Abacus: A Service-Oriented Programming Language for Grid Applications," presented at The 2005 IEEE International Conference on Services Computing (SCC 2005), Orlando, Florida, USA, 2005.

[4] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft, "SmartFrog: Configuration and Automatic Ignition of Distributed Applications," 2003 HP Openview Univ. Assoc. Workshop, Hewlett-Packard Labs, 2003.

[5] T. F. M. Smith and B. Freisleben, "Hot Service Deployment in an Ad Hoc Grid Environment," presented at Proceedings of the 2nd Int. Conference on Service-Oriented Computing (ICSOC'04), New York, USA, 2004.

[6] I. Foster, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, H. Kishimoto, F. Maciel, A. Savvy, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. v. Reich., "The Open Grid Services Architecture, Version 1.0," https://forge.gridforum.org/projects/ogsa-wg/document/draft-ggf-ogsa-spec/en/, 2004.

[7] S. Viswanatham and N. Islam, "Managing and Monitoring Web Services in Project GlassFish," http://developers.sun.com/prodtech/appserver/reference/techart/ws_mgmt.html, March 1, 2006

[8] "JSR-181: Web Services Metadata for the Java Platform,"http://jcp.org/en/jsr/detail?id=181, June 1, 2005.

[9] "JSR 175: A Metadata Facility for the JavaTM Programming Language," http://jcp.org/en/jsr/detail?id=175, September 30, 2004.

[10] P. Leach, M. Mealling, and R. Salz, "A UUID URN Namespace," http://www.ietf.org/internet-drafts/draft-mealling-uuid-urn-05.txt, vol. draft-mealling-uuid-urn-05 (work in progress), 2004.

[11] "AVM," http://vega.ict.ac.cn/gos/avm/avmdoc.htm.

[12] M. A. Frumkin and R. F. V. d. Wijngaart, "NAS Grid Benchmarks: A Tool for Grid Space Exploration," Cluster Computing, vol. 5, pp. 247 - 255, 2002.

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L.Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "THE NAS PARALLEL BENCHMARKS," NAS Technical Report RNR-9 1-002, NASA Ames Research Center, Moffett Field, CA, 1991.