A Service-Oriented Virtual Machine for Grid Applications

Hong Liu\(^1,2,3\) Wei Li\(^1,3\) Xiaoning Wang\(^1,2,4\) Yili Gong\(^1,2,3\) Tian Luo\(^1,2,4\)
\(^1\) (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)
\(^3\) \{liuh, liwei, gongyili\}@ict.ac.cn \(^4\) \{wangxiaoning, luotian\}@software.ict.ac.cn

Abstract

Grid computing is a new paradigm for distributed computing, and service has become building block of grid applications. However, current approaches cannot free developers from low-level laborious work when building grid applications. We propose a service-oriented virtual machine called Abacus Virtual Machine to simplify the task of grid application development. As a language level virtual machine, it provides a service-oriented instruction set to abstract the operations on the services of a grid application. It also virtualizes services and creates a virtual global system image for grid applications, thus services can be transparently distributed and shared. In this way, Abacus Virtual Machine hides the cumbersome underlying details from programmers and reduces the complexity greatly in grid application development.

1. Introduction

Web Service [12], which is designed to support interoperable machine-to-machine interaction over network, has been an important enabling technology for grid communities. Service-oriented architecture (SOA) has been used in several implementations of grid middleware [6] [11]. Thanks to their effort, grid technologies become increasingly popular.

However, the way to develop and manage services in grid applications is still inefficient. First, most current programming environments don’t provide a language-level abstraction for services, and the developers have to consider the low-level details when building grid applications, such as how to distributed services in the network and how to deploy them. Second, many jobs of service management, e.g. service deployment, are handled manually. It’s difficult to deploy a service on-demand at run-time and allocate resources to it dynamically. Third, when a service’s network address is changed, the service’s endpoint should also be changed, which is impractical to ask the users to manually handle these changes, such as changing the codes of their programs. In order to solve the above problems, following requirements should be satisfied:

- **High-level abstracts.** For the programmers, a small number of concepts and programming interfaces can ease grid application development. Common functions of Grids should be abstracted at a higher level independent to low level details such as platforms, protocols and managing schemes. Programmers can concentrate more on application logic, which makes application more portable. Furthermore, for the run-time environment, high-level abstraction is the base for further optimization.

- **Automatic service management.** Currently, grid programmers labored over handling of service management tasks such as service deploying, discovering, destroying, and etc. However, in an ideal way, programmers only need to claim the services they need, and the underlying systems should automatically perform the actual management work.

- **Transparent sharing among grid applications.** In conventional systems, code and data sharing techniques such as shared library, components, and shared objects, greatly ease the software developing. These mechanisms support the *implicit sharing*, which means that sharing occurs at compiling and executing time but not be written in programs by programmers. Due to the dynamic and autonomous nature of Grids, similar mechanisms should also be provided to facilitate resource sharing among different applications. The underlying system should support the loading and
linking of program components distributed in different sites.

In this paper, we propose a novel approach that is based on a virtual machine which is called Abacus Virtual Machine (AVM). AVM is a hosting environment for grid computing where service acts as a basic entity. AVM provides a service abstraction and the operations on services are abstracted to a service-oriented instruction set, hence a service can be introduced as a first-class object in the programming language. Because there isn’t any corresponding abstraction in existing programming languages yet, we propose a high-level programming language, Abacus [20], which supports developers to build grid applications with the service abstraction. Furthermore, to provide a virtual global system image and completely decouple grid applications and underlying physical network, we create a unified virtual service address space layer.

The rest of the paper is organized as follows. Section 2 introduces our design principles. Details of AVM system are presented in Section 3. We present the results of our evaluation in Section 4. Section 5 and 6 provide the related work and concluding remarks respectively.

2. Design principles

In our approach, the key concept is Abacus service, which is the first class object in our design. The first class means that in AVM a service can be expressed, created, passed, invoked, and destroyed at language level, which is different from the Application Programming Interface (API) approach in current solutions such as Java [7], AXIS [2] and GT4 [6]. The design principles of Abacus service can be summarized as follows:

- **Abacus service should be a manageable entity.**
  The benefit is that AVM can manipulate every step of service in its full life-cycle and provide system functions such as service deployment, type checking, garbage collection and exception handling. Compared with AVM, current service-oriented environments only provide partial management of services or leave it completely to programmers.

- **Abacus service should be a global addressable entity.**
  Compared with object in object-oriented systems, service has an inherent nature that it should be accessible to others, which needs a global address space model. Conventional virtual machines use private address space model, for example, in JVM, if two Java applications at different site need to share resources, they have to use message passing mechanism such as RMI [18] to exchange data among different spaces.

3. Abacus Virtual Machine Design

3.1. System Overview

AVM is a high-level-language virtual machine providing an abstracted service-oriented infrastructure for grid application. In AVM, service is a machine abstraction, and a grid application can be regarded as a collection of autonomy services that coordinate with each other. Developers write grid applications in Abacus Language and compile them to bytecodes which contain instructions for AVM. Then users can run these applications on AVM.

![Figure 1. The AVM System](image)

As shown in Figure 1, each AVM is a daemon program running on an operating system which hosts services for Abacus applications. In this way, an AVM can be viewed as a Local Service Container (LSC). When a client request arrives, AVM will find the target service in LSC and take advantage of the local resources to perform requests. Meanwhile, AVM is a distributed system with a uniform global view of services. All LSCs constitute a Global Service Container (GSC). In GSC, each service can access any other services directly. One AVM can deploy services on another AVM and visits them by their address in GSC.

3.2. Abacus Programming Language

To provide service abstraction in the language level, a service-oriented programming language named Abacus is designed as a developing tool. Details of the language can be found in [20].

In Abacus, the primary operations of a service include service deploying, service binding, and service invoking.

Respectively, the service deploying can be done through the following statement:
NewExpression: 
  New ServiceName();

This statement deploys a service and returns its virtual address, an internal identifier of the service. At the same time, it implicitly binds the application with the service just deployed.

The syntax of service invocation is similar to the method invocation in Java Language:

MethodCall: 
  ServiceVar.InterfaceName(parameterlist);

3.3. Address Space Model

In AVM, grid applications create, use and destroy services in the global service space at run-time. Thus, how to design a suitable service space to locate and organize services is an important issue in AVM design.

Figure 2 [20] gives the two-level address space model in AVM. The bottom level is a URL-based address space, where each service is a standard web service addressed by a uniformed resource locator (URL). It is called physical service space (PSS), for it is managed by autonomous physical sites (i.e. local service containers).

![Abacus Address Space Model](image)

The upper level is a UUID-based space. In this space, a service can be addressed by its virtual address, which is a 32-byte long Universally Unique Identifier (UUID) [13]. This space is called virtual service space (VSS), for it is maintained implicitly by Abacus compilers and AVM, and PSS is mapped to this space at run-time. One role of VSS is to provide a global uniform view shared by all AVMs. If it’s permitted, an Abacus program can directly access other program’s service by its virtual address without third-part interventions. Another role of VSS is to provide a location-independent feature for Abacus programs. When a program moves from one site to another, other programs can still access their services by virtual addresses. In Fig. 2, when program Y moves from web site B to C, program X can still access the services provided by Y by their virtual addresses.

The virtual address of a service is created at compiling time or runtime. At the time of compiling, Abacus compilers assign a 32-byte long virtual address for a global service variable, which can be linked to client programs as libraries. At runtime, the services created dynamically are also assigned with virtual addresses dynamically.

4. Evaluations

In this section, we present the experimental programs and the results based on these programs. The goal is to answer the question of whether AVM is good enough for building service-oriented applications. However, the question itself is rather general and hard to answer. Accordingly, we rely on three, more specific criteria and corresponding questions to evaluate our system:

- **Usability** Can we build useful programs using Abacus language and AVM? This criterion determines whether our architecture is capable of supporting service-oriented applications.
- **Complexity** How hard is it to develop application in AVM? This criterion determines the effort involved in developing programs on our architecture.
- **Performance** Is system performance acceptable? This criterion determines whether our architecture performs well enough.

To summarize the results, we show that AVM (1) is powerful to support web service-oriented programs on top of it, (2) is easier to program than with conventional programming styles, (3) has acceptable performance. In other words, our experimental results show that AVM system does provide a service-oriented infrastructure to simplify the task of programming grid applications.

AVM is built on the top of the freely available Kaffe virtual machine [8]. It currently runs under Linux on X86 as well as on ARM9. The measurements reported in this section were performed in LAN as well as WAN. The LAN testbed consists of six PC servers with dual Intel Xeon 2.4GHz processors, 1GB of RAM. The servers are connected by a 100 Mb Ethernet. The operating systems on these servers are Redhat Linux with kernel 2.4.18. The WAN test environment includes three nodes of China National Grid located in different cities.

4.1. Usability

To evaluate our architecture, we built a set of web service-oriented programs. In this section, we present a
standard benchmark program, NGB, and implements it in the AVM platform.

We use NGB (NAS Grid Benchmarks) [15], a benchmark suite for grid computing, to measure the usability and performance of AVM. NGB is evolved from NPB (NAS Parallel Benchmarks) [4], which is widely used on parallel computing systems. In NGB, there are five basic tasks (i.e. BT, FT, LU, MG, SP) on behalf of various type of scientific computation, and these tasks construct four problems representing four typical classes of grid applications, named respectively Embarrassingly Distributed (ED), Helical Chain (HC), Visualization Pipe (VP), and Mixed Bag (MB). 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 tasks have different amounts of computational work.

From our practice, we’ve found these benefits brought by AVM, which can be summarized as follows,

- **Quick development** Comparing to other developing tools, AVM eliminates the cumbersome details and simplifies the task of programming grid applications. We’ll discuss this aspect in detail in the next section.
- **Easy to manage** The service in AVM can be created, deployed and destroyed at run-time. So in AVM, we needn’t manually deploy services on different servers. Instead, we just wrote a simple service called “register center” and the register center was able to deploy and destroy services according to the application’s running states.
- **Adaptive to network environment** The virtualization of service hides service’s location information. Thus, when we ran NGB on various occasions, we needn’t reconfigure the application to adapt to the various network environments. While on Tomcat/Axis platform, it’s troublesome to reconfigure the application by changing both source codes and WSDD files.

4.2. Complexity

To evaluate the effort involved in writing adaptable applications, we analyze the two aspects of the process of implementing and running NGB: development steps and code length. The system we choose to compare with AVM is Java/Axis/Tomcat, one of the most popular hosting environments for grid applications.

In Fig. 3, we’ve summarized the development steps of the two environments.

![Figure 3. Comparison on development steps](image)

Using AVM and Abacus, programmers do not have to write or revise a WSDD file for their services, or deploy services manually. In addition, programmers do not have to hold detail knowledge at low-level such as WSDL, XML, and WSDD.

### Table 1. Comparison of Code and Configuration Lines

<table>
<thead>
<tr>
<th></th>
<th>Code lines</th>
<th>Configuration Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomcat/Axis</td>
<td>2424</td>
<td>26 lines for the server and 34 lines for client.</td>
</tr>
<tr>
<td>AVM</td>
<td>2401</td>
<td>0</td>
</tr>
</tbody>
</table>

To quantify the effort involved in building NGB, we measure the code and configuration lines of the two implementations of NGB in Table 1. The code length of Abacus is about 15% less than the code length of Java. It is mostly because of the simplification of service processing codes. For example, lines of one service invocation are reduced from at least 8 lines in Java to only one line in Abacus. In addition, in AVM programmers need not write any configuration file, while in Tomcat/Axis, programmers should write 60 lines in configuration files.

4.3. Performance

Because AVM creates a new abstraction layer for grid applications, it’s inevitable to add some overhead. In addition, choosing SOAP as an interface method also decreases the system’s performance. To quantify the influence of the overhead, we measured performance of primitive operations and NGB test program. We choose two comparable systems: Tomcat/Axis, another platform to host web service, and Java/RMI [18], a distributed system using
appropriative interfaces. From the following tests, we can see that the overhead brought by the virtualization is acceptable and AVM reaches a fairly good performance.

4.3.1. Primitive Operations. In this section, we tested and compared the time spent on primitive operations on Kaffe (Java/RMI), AVM (service) and Tomcat/Axis. The primitive operations include creation and invocation of a remote object and a service. Table 2 shows the comparison between primitive operations in these three environments.

<table>
<thead>
<tr>
<th></th>
<th>New/deploy</th>
<th>Invoke-LAN</th>
<th>Invoke-WAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaffe</td>
<td>2us</td>
<td>1.35ms</td>
<td>159ms</td>
</tr>
<tr>
<td>AVM</td>
<td>45.575us</td>
<td>1.49ms</td>
<td>170.28ms</td>
</tr>
<tr>
<td>Tomcat (manual)</td>
<td>301.05ms</td>
<td>520.28ms</td>
<td></td>
</tr>
</tbody>
</table>

From Table 2, we can see that AVM spends more time on these primitive operations than Kaffe, for it has to map the virtual address to the physical address. In addition, the SOAP protocol used in AVM costs more time than RMI in Kaffe.

While on Tomcat, the time spent on invocation is far greater than other two platforms. There are two reasons accounting for this. First, Tomcat is not a special container designed for web-service applications. It performs some unnecessary operations, such as logging, etc., other than only processing the SOAP message. Second, the container and SOAP processing modules are implemented in JAVA. It is apparently slower than the embedded container in AVM.

4.3.2. NGB Test. The above measurements show that the performance of primitive operations on AVM is slightly lower than the corresponding operations on Java/RMI, while is better than the operations on Tomcat/Axis. Because the programs running on Java/RMI have the smallest overhead in communication and the best performance, we use their results as reference marks and compare the results on AVM with them.

We respectively executed five basic tasks of NGB on AVM and on Kaffe (in Fig. 4). Each task presents a kind of scientific computation separately. Because of the relative large overhead of service management (service vs. object) and communication (SOAP vs. RMI), the MG and FT tasks on AVM are about 20% slower than that on Kaffe. However, it’s a bit out of expectation to see that the SP, LU and BT tasks run fast on AVM than on Kaffe. The reason is as follows. These three tasks have a common feature that they have a huge number of operations of accessing instance variables. In Kaffe, the inheritance and polymorphism properties increase the cost of accessing variables. On the contrary, the AVM takes away inheritance and polymorphism features. As a result, these tasks can be executed faster on AVM than on Kaffe.

5. Related Work

Due to the advantage of security, isolation, resource control and site-independence [14], virtual machines are attracting increasing attention in distributed computing. The Xenoserver project [19] introduces a distributed infrastructure based on the Xen virtual machine [3]. On the other hand, the Virtuoso [17] and Violin [21] projects explore networking issues related to using virtual machines in a Grid environment. Besides, the globus alliance released the virtual workspaces project [9], which provides an abstraction of an execution environment that can be made dynamically available to authorized clients by using well-defined protocols.

Although they are all virtual machines in distributed computing environment, AVM is a high-level-language virtual machine that focuses on simplifying the development of web service-oriented grid applications; on the contrary, the VM systems mentioned above are engaged in providing an execution environment for the existed grid applications. The grid middleware technologies, such as GT4 and OMII, are designed to enable applications that federate distributed resources. Unlike AVM, these systems do not make a new developing environment but provide toolkits for the traditional environment.

6. Conclusions and Future Work

In this article, we have explored an innovative service hosting environment for grid applications. Firstly, our system provides a core abstraction, Abacus service. Consequently it hides the low-level details of services and automatically manages services. Secondly, the service virtualization makes services adaptive and
easy to share in a dynamic environment. Even the service’s location is changed, the users don’t have to reconfigure the system or change their codes at all. According to results of the results of evaluation programs, AVM does simplify the task of building grid applications. Furthermore, the overhead brought by the virtualization is acceptable.

We have presented a first cut at a service-oriented virtual machine. On the base of AVM system, many improvements are still required to be done. In the future, we will provide new features such as dynamic service scheduling, which can immigrate service to other AVM while one AVM is busy; error recovery, which can deploy a new service while one service has corrupted etc. These improvements will make our system more flexible, adaptive and stable. In addition, the way of communicating with external services is still unsatisfactory and porting a middleware system to AVM environment are also parts of our future work.

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

[9] K. Keahey, I. Foster, T. Freeman, and X. Zhang, Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid, Accepted for publication in the Scientific Programming Journal, 2006