

# Provisioning for Dynamic Instantiation of Community Services

On-demand provisioning can allow collaborative communities to rapidly deploy the services required to support collaboration, without the need to acquire and operate dedicated hardware. To meet community needs for on-demand access while also maximizing global availability and runtime efficiency, the authors propose service-, container-, node-, and VO-level provisioning approaches based on a highly available dynamic deployment infrastructure. Their experiments with an image-processing application demonstrate their approach's efficiency and effectiveness.

The grid is an Internet-connected computing environment in which computing and data resources are located in different administrative domains, often with distinct security and resource utilization policies. The Open Grid Service Architecture (OGSA)<sup>1</sup> has moved grid computing's focus from legacy, computing-intensive applications to service-oriented computing (SOC)<sup>2</sup> based on open standards. Globus Toolkit (GT)<sup>3</sup> development has followed this trend, with GT4 building on the Web Service Resource Framework (WSRF) specifications to provide an efficient, extensible, stateful, and flexible grid middleware. Dynamic instantiation of services in a WSRF-enabled container can provide high availability during runtime and flexible automation for grid applications.

Many investigations have addressed and tracked this trend.<sup>4,5</sup>

Another important concept in grid computing is the *virtual organization* (VO).<sup>6</sup> A VO aggregates services and computing resources to meet the needs of a distributed user community. VOs are usually dynamic, meaning not only that membership might vary over time but also that VO services – and those services' computing requirements – can change as the VO's size and function evolve. Thus, at various stages during a VO's life cycle, the set of resources it uses can vary – for example, in response to changing quality-of-service demands for existing services, or a need to instantiate entirely new services.

This need for on-demand computing motivates our interest in provisioning. More specifically, VOs seek to improve

**Li Qi and Hai Jin**

*Huazhong University of Science and Technology*

**Ian Foster and Jarek Gawor**

*Argonne National Laboratory*

## Related Standards

○ OASIS released a standard that provides an XML framework for managing the provisioning and allocation of identity information and system resources within and between organizations: the Service Provisioning Markup Language (SPML; [www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=provision](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=provision)). The SPML model consists of four main roles: requesting authority, provisioning service provider, provisioning service target, and provisioning service objects. It defines the communicating protocols between these roles.

According to the requirements from grids and virtual organizations, the Configuration, Description, Deployment, and Lifecycle Management (CDDLDM; <http://forge.gridforum.org/projects/cddl-dm-wg>) standard that the Open Grid Forum (OGF) proposed standardizes distributed software deployment and configuration in a validated life cycle.

The Application Contents Service (<http://forge.ogf.org/sf/projects/acs-wg>) in OGF aims to establish a standard interface for storing and exchanging archives of application contents, defining both the Application Repository Interface (ARI) as an open grid service architecture (OGSA) service and the standard Application Archive Format (AAF). The application contents services will promote efficient and automated deployments in grid systems and interoperability among grid implementations.

Finally, to define an on-demand provisioning interface via Web services, OASIS is developing the Web Services Distributed Management (WSDM; [www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=wsdm](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm)) specification, which focuses on two distinct tasks: management using Web services and management of Web services.

As a supplement to these standards, our work addresses runtime capabilities during the instantiation procedure for VO services.

scalability, flexibility, and availability for services hosted on a provisioning-enabled grid. Scalability lets services adapt to changes in VO scale, such as the number of users. Flexibility allows users from different VOs to easily reconfigure, redeploy, and undeploy services without shutting down the whole system. High availability means that services will function correctly and continuously despite faults such as network congestion, service crashes, and maintenance.

Here, we describe mechanisms for on-demand provisioning that we have implemented in our Highly Available Dynamic Deployment (HAND) infrastructure.<sup>5</sup>

### Concepts

Before we discuss our provisioning solution's design and implementation, let's look at some basic concepts related to VO provisioning:

- A *container* in a Web services-based system such as GT4 hosts services and executes cli-

ent-issued user requests that invoke the operations that those services define.

- A *service* is composed of an interface – defined in terms of standards such as SOAP, the Web Services Description Language (WSDL), and WSRF<sup>2</sup> – and an implementation, which might comprise functions, configurations, and optional supporting software libraries. Grid developers can easily compose new services with existing services to realize new functions.
- *On-demand provisioning* means defining a VO's configuration according to organizational requirements. That is, according to system utilization and service-level agreements, the infrastructure will deploy (or undeploy, redeploy, or reconfigure) related ones automatically and dynamically.
- *Dynamic instantiation* means that remote clients can request that a VO application upload, deploy, instantiate, and activate new services into – or deactivate, destruct, and undeploy services from – existing containers according to user demand or VO application status. The activation (or deactivation) operation inside a container makes a service instance ready (or unready) to perform its function using the correct resource states.

To let VO applications share grid resources concurrently and efficiently, the containers on those resources must support dynamic instantiation in order to maintain the services and further achieve on-demand provisioning.

### On-Demand Provisioning Scenario

To illustrate important requirements for on-demand provisioning in VOs, consider the following scenario. Jack, a member of the Image Processing Virtual Organization (IPVO), issues a request to a service, the Skeleton-Transformer, that hasn't yet been deployed on any physical resource. A single request to this service can use many nodes concurrently, as long as the Skeleton-Transformer application has been instantiated on those nodes.

Skeleton-Transformer service deployment proceeds as follows. First, a provisioning module analyzes resource requirements, identifies suitable resources, and works out what the grid infrastructure must do to deploy the service to these resources. Unfortunately, another VO,

CFD Computing (CFDVO), is already using the selected resources – Mary, a CFDVO member, submitted a request several days ago that's still running. In principle, IPVO and CFDVO can share resources, but in practice, deploying the Skeleton-Transformer service while Mary's job is running will cause Mary's job to fail because it must share key infrastructure components with IPVO. Hence, Jack must either change his requirement or wait for Mary's job to finish; we would prefer to avoid the latter.

After a long waiting period, the provisioning module begins to propagate the service implementation to the selected resources. However, to correctly instantiate this service, the infrastructure must first update various services already deployed on those resources. We want to be able to detect a service's status consistency on resources in advance, so that we can take this information into account when selecting resources.

In addition to these problems, when the computing resources inside a local VO aren't enough to allocate for the job, Jack must also consider how to lease the resources from other VOs safely and efficiently. Ultimately, deployment completes, but only after substantial time delay and considerable user involvement.

## Design and Implementation

From Jack's experiences, we conclude that on-demand provisioning should address the following challenges:

- *Efficiency.* During dynamic runtime, the infrastructure should quickly propagate the instantiation of requested services to specific VOs in parallel approaches.<sup>7</sup> Doing so can enhance VO applications' availability and reduce downtime.
- *Service isolation.* The infrastructure should isolate services hosted on the same physical resource, so that provisioning or maintenance of one service doesn't affect others.
- *Correctness.* To provide correctness in provisioning, the infrastructure should resolve two issues. First, it must deal with dependency complexity – if the infrastructure can detect dependencies between service components, environments, and instructions before provisioning, we can avoid many mistakes.<sup>8</sup> Second, it must manage consistency. Many emergencies might disturb provisioning on

distributed resources. Inconsistent states of provisioning due to these emergencies will result in later requests failing. To achieve high availability and correctness, we need a mechanism to guarantee the maintenance of dependencies and consistencies among all leased resources in the infrastructure.

- *Security.* On-demand provisioning might bring unpredictable crises to VO resources (for example, requests to deploy Trojan viruses), so the infrastructure should constrain different users' privileges and guarantee safety for different VO resources. Furthermore, it should provide a channel that will let alien VO users access local resources for higher utilization.

These challenges motivated us to design our provisioning solution with four levels: service, container, node, and VO, respectively.

### Service- and Container-Level Provisioning

Service-level provisioning mainly resolves the service isolation issue. By enhancing such provisioning, a service that the infrastructure is deploying or undeploying is deactivated and then reactivated, leaving all other services unaffected.

With container-level provisioning, on the other hand, the infrastructure reloads the entire container when issuing the provisioning – that is, all services in the container are deactivated and reactivated. We discuss this level of provisioning more in previous work.<sup>5</sup>

### Node- and VO-Level Provisioning

Our motivation for designing node-level provisioning was to orchestrate provisioning efficiently, correctly, and safely in an internal VO. It mainly resolves the dependency complexity issue. By maintaining a *dependency map* in the provisioning center, node-level provisioning can expand the possible provisioning task automatically, according to dependencies, and then orchestrate requests to target resources according to a specific propagation policy.

Figure 1a illustrates a complete provisioning procedure in VO1. When a client  $c_1$  requests a service  $s_1$ , the provisioning center determines that the correct deployment of  $s_1$  requires that  $s_2$  be deployed first. Moreover, it detects an inconsistency:  $s_2$  is already deployed on the candidate resource  $r_2$  (and thus need not be deployed on that resource). Following this analysis and necessary authorization, the execution planner

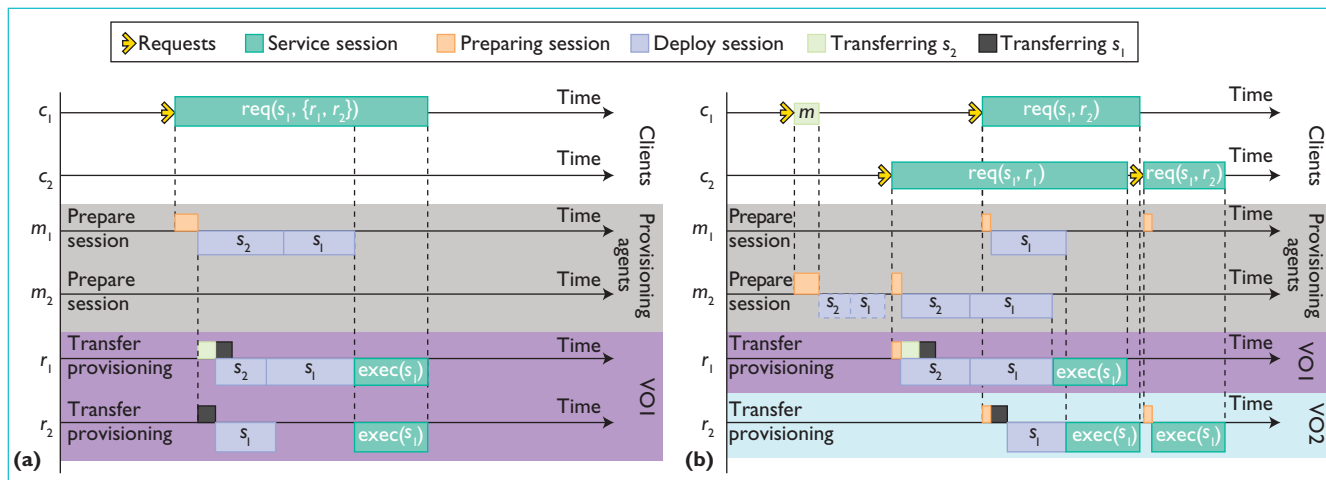


Figure 1. Node-level vs. virtual organization- (VO-) level provisioning. (a) Node-level instantiation focuses on dependency complexity and fast propagation inside a VO equipped with homogeneous resources. (b) VO-level instantiation, on the other hand, focuses more on security and high consistency between different VOs.

generates an appropriate workflow, which it delivers to the provisioning agent  $m_1$ , which then transfers related service packages to targets  $r_1$  and  $r_2$ , respectively. After provisioning, the client  $c_1$  executes the requests correctly.

Compared with node-level provisioning, VO-level provisioning is more concerned with security challenges and the high inconsistency problems that occur in alien VOs. Because they are autonomous, VOs always employ independent security policies to ensure security. Achieving runtime instantiation of services in alien VOs requires infrastructure support. We adopted a delegation approach to crossing VO accesses by authorizing limited privileges to other alien VOs. In addition to security, different VOs vary considerably due to different functionalities – for example, because the CFDVO service mentioned in our scenario is compute-intensive, it will prefer computationally powerful resources. In contrast, the data-intensive IPVO application prefers resources with a high-bandwidth network. If provisioning between the VOs is synchronized, it will lower server usage, so we adopted an asynchronous provisioning model for VO-level provisioning. HAND won't execute the physical provisioning immediately – rather, it will match the provisioning states and execute physically when a user requests the service.

The example in Figure 1b describes the procedure. We use a similar case in Figure 1a, except resource  $r_2$  here is occupied by VO2. Obviously, when client  $c_2$  in VO1 requests an internal service, the procedure is similar to node-level pro-

visioning. Once the provisioning center delivers  $c_1$ 's request  $req(s_i, r_2)$  to the alien VO, it then dispatches the request to agent  $m_1$ , who has received authorization to deploy service  $s_i$  in VO2. After  $m_1$  finishes deploying the service  $s_i$  in VO2,  $s_i$  will correctly execute the later requests from VO1. Figure 1b also depicts inconsistent provisioning: even though  $c_1$  requested the provisioning of  $m$  earlier, the real provisioning happens asynchronously, according to respective situations.

## Implementation

To orchestrate the four levels of provisioning effectively and efficiently, we implement the provisioning architecture and infrastructure based on the GT4.0 Java Web service core. As Figure 2a shows, we view the provisioning as an infrastructure that fuses various components. The four levels of provisioning are distributed across different layers – container- and service-level provisioning are both implemented as part of the dynamic deployment infrastructure on the GT4 Java Web service core, whereas node- and VO-level provisioning are implemented in the provisioning center.

Because the provisioning center's architecture is centralized, we use a multiagent technique to achieve higher throughput. The agents deliver provisioning operations to target resources in the VOs. Similarly, we implement the VO monitor as a distributed module for different sized VOs to improve throughput.

Figure 2b presents a sequence diagram of the HAND system. At the instantiation phase, the VO application demands that the service  $S_a$  be

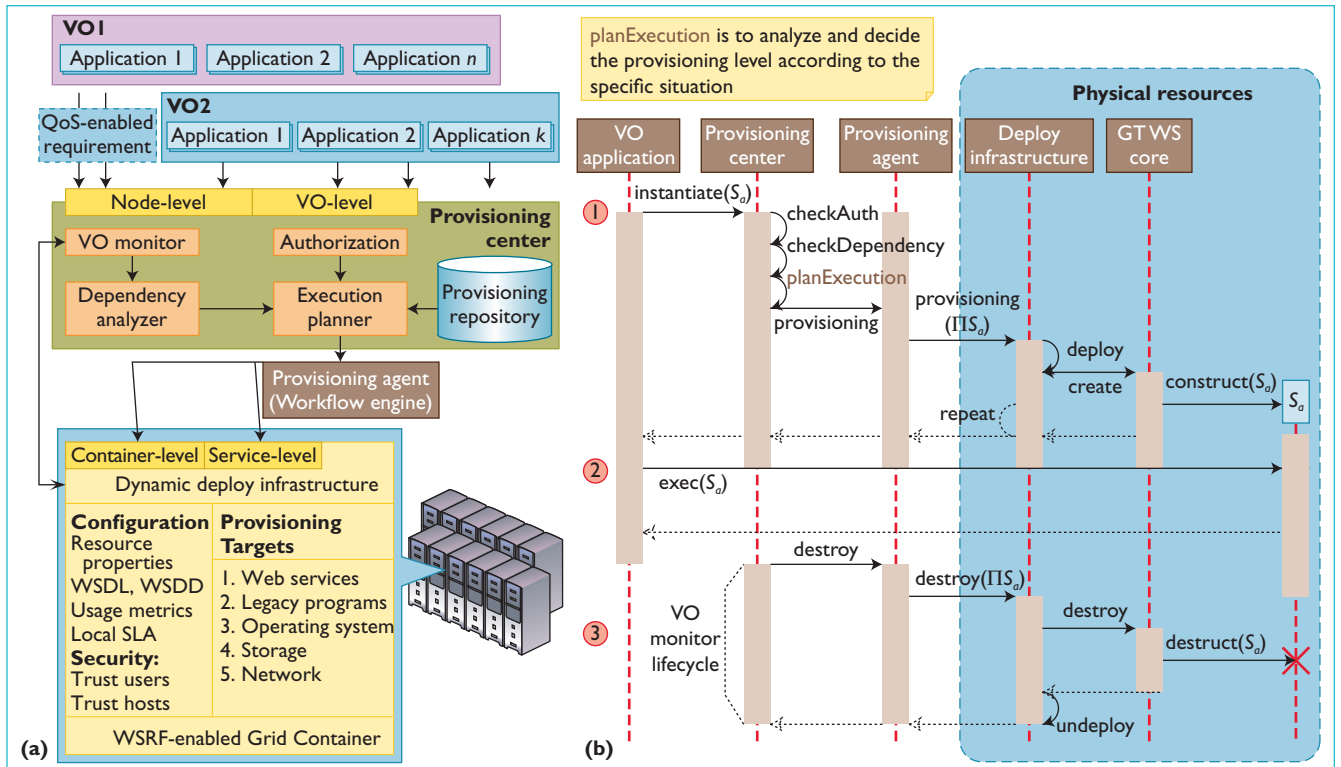


Figure 2. On-demand provisioning architecture and its working flow. (a) The provisioning system includes resources with a dynamic deployment infrastructure, provisioning center, provisioning agents, and various virtual organizations (VOs). (b) The provisioning procedure can be decoupled into three phases: instantiation, execution, and destruction.

instantiated on several resources. In particular, the *authorization module* checks the authentications; the *dependency analyzer* mines possible dependencies ( $checkDependency$ ); and the *execution planner* adopts a suitable provisioning level and generates the provisioning task for all related services ( $IIS_a$ ) to provisioning agents. By adopting a suitable provisioning level, these agents invoke the remote deploy service to issue physical instantiations on specific containers. Furthermore, the leased container returns the  $S_a$ 's end-point references to the VO application, which then delivers the requests to the instances so they can finish the job in the execution phase. Finally, the *VO monitor* automatically destructs instances of service  $S_a$  on resources and undeploys them dynamically in the destruction phase, according to the  $S_a$  instance's life cycle. If domain users prefer manual management, they can also explicitly destroy those service instances by invoking the undeployment interface.

### Adopting a Provisioning Level

The execution planner determines the appropriate provisioning level by analyzing user demands, available resources' runtime status, and service-

level agreements. Specifically, when the number of related services in a single container is less than a threshold  $q$ , the execution planner will adopt service-level provisioning; if the number of related services is greater than  $q$ , it will adopt container-level provisioning. We evaluate the value of  $q$  according to the historical practices in leased resources<sup>5</sup> before provisioning. When the instantiations can run in parallel according to the dependency analyzer's evaluation, the execution planner adopts node- and VO-level provisioning, respectively, to invoke the lower-level provisioning. It decides which to adopt based on the resources available to the VO. Namely, the execution planner adopts VO-level provisioning when the VO user needs to access alien VO resources; otherwise it adopts node-level provisioning. Naturally, the VO user can directly specify preferred approaches in order to finish instantiations for each application's specific nature.

### Metrics

To understand the capability of provisioning qualitatively and quantitatively, we measure it via the concept of self-maintainability. By adopting a provisioning approach with high

## Related Work in On-Demand Provisioning

Many researchers have explored and developed on-demand provisioning in different contexts, including in Java 2 Enterprise Edition (J2EE)<sup>1</sup> and Web services.<sup>2</sup>

The Globus Workspace Service that Kate Keahey and colleagues propose<sup>3</sup> uses virtual machine technology (such as Xen and VMware; <http://workspace.globus.org/vm/index.html>) to build virtual working environments and enable virtual organization (VO) users to dynamically manage the grid job life cycle. Their focus is on operating system (OS) image deployment rather than service instantiation.

IBM's Tivoli Provisioning Manager (TPM)<sup>4</sup> is built on service-oriented architecture principles. It enhances usability for executing changes while keeping server and desktop software compliant. TPM helps organizations with provisioning, configuration, and maintenance of servers, virtual servers, and other resources. It supports OS, software, and storage provisioning.

Hewlett-Packard's Smart Frog<sup>5</sup> is a framework for service configuration, description, deployment, and life-cycle management. It consists of a language for describing component collection and configuration parameters, a deployment infrastructure that activates the application description, a component model that manages the components to deliver and maintain running systems, and a set of components that provides various application services.

The SUN NI Service Provisioning System (SPS)<sup>6</sup> automates the deployment of multitier applications across heterogeneous resources. A highlight of SPS is that it can simulate the deploy-

ment process on target systems prior to actually implementing the changes, which ensures successful delivery. In addition, version control and role-based access control can help users construct applications efficiently.

Our approach is distinguished by its focus on the instantiation of VO services rather than OS or related workspace software, runtime availability during service-oriented provisioning, and provisioning granularities from service- to VO-level provisioning, which satisfies demands from different VOs.

## References

1. F. Reverbel, B. Burke, and M. Fleury, "Dynamic Deployment of IIOP-Enabled Components in the JBoss Server," *Proc. 2nd Int'l Working Conf. Component Deployment* (CD 04), Springer-Verlag, 2004, pp. 65–80.
2. P. Watson and C. Fowler, *An Architecture for the Dynamic Deployment of Web Services on a Grid or the Internet*, tech. report series CS-TR-890, Univ. of Newcastle upon Tyne, 2003.
3. K. Keahey et al., "Virtual Workspaces in the Grid," *Proc. European Conf. Parallel Computing 2005* (EuroPar 05), Springer-Verlag, 2005, pp. 421–431.
4. D. Chess et al., "Experience with Collaborating Managers: Node Group Manager and Provisioning Manager," *Proc. 2nd IEEE Int'l Conf. Autonomic Computing* (ICAC 05), IEEE CS Press, pp. 39–50.
5. B. Agarwalla et al., *Automating Provisioning of Complete Software Stack in a Grid Environment*, tech. report GIT-CERCS-04-26, Center for Experimental Research in Computer Systems, 2004.
6. K. Zielinski, M. Jarzab, and J. Kosinski, "Role of NI Technology in the Next Generation Grids Middleware," *Proc. European Grid Conf. 2005*, Springer-Verlag, 2005, pp. 942–951.

self-maintainability, administrators and users from different VOs can benefit from fewer interactions and higher transparency. Specifically, the qualitative measures include the following:

- The ability to issue dependencies and consistency. A self-maintainable provisioning system should shield different VO applications from the inconveniences of complicated inconsistency and dependency. Instead, it gives them common and simple interfaces. Users need not consider dependency and consistency among VO resources.
- An automated maintenance or provisioning process. According to the runtime status, a self-maintainable infrastructure can generate provisioning workflow according to multiple users' requests automatically.

In addition to these two qualitative measures, self-maintainability includes a set of quantitative metrics:

- Provisioning time speedup. We take the provisioning of a single service component at the container level as our standard reference. We define the speedup  $s$  as the ratio of the sum of provisioning time  $t_i$  for the provisioning task set  $P$  at the container level to the actual provisioning time  $T_s$ :

$$s = \frac{\sum_{i \in P} t_i}{T_s}$$

- The global system's availability during provisioning. This metric measures the ratio between the number of requests executed correctly and the total number of requests delivered during the maintenance procedure.

By comparing the provisioning solutions with the self-maintainability metrics, VO users can easily find out which is the best one for different VO applications.

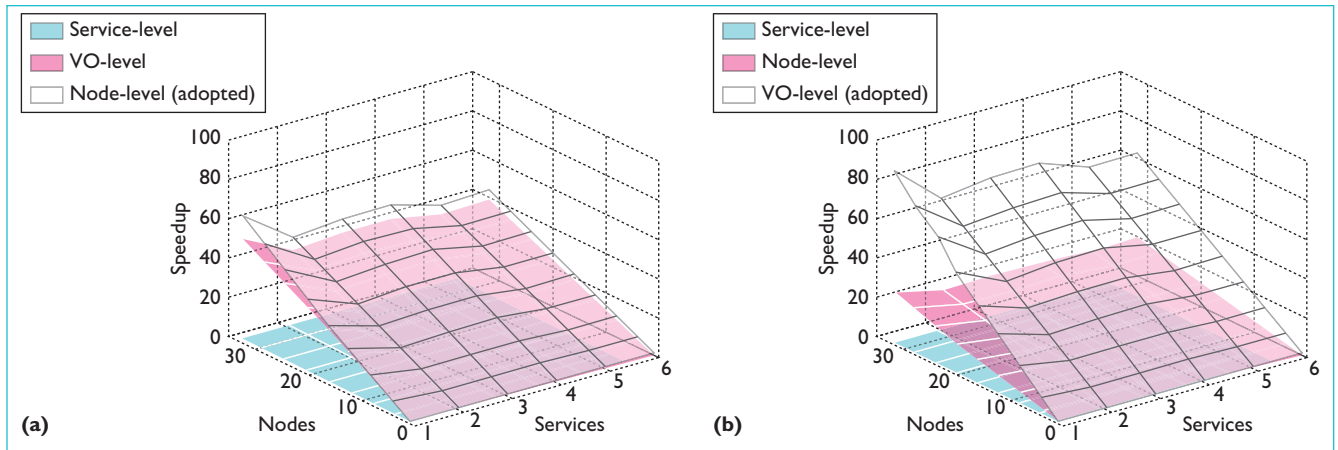


Figure 3. Speedup as a function of provisioning scale and related service sizes. We measured performance using (a) the HUST-VO resources only and (b) when also using the resources from two virtual organizations. We compared all results to container-level provisioning.

## Experiments and Qualitative Comparison

To evaluate the performance of our different provisioning methods, we executed a real-world image-processing application, called Spot Detector, in the ChinaGrid testbed,<sup>9</sup> which is distributed over 22 major universities in China and comprises five application grids, each constructed from several VOs to serve various scientific projects. We deployed the HAND infrastructure on testbed resources. The adopted application, Spot Detector, can use several processors in parallel when responding to a client request to identify a supplied mark in a large remote sensor map. Thus, allocating additional service instances can reduce processing time. However, to provision a new service instance, the VO user must deploy six dependent services first. Correctly handling these dependencies requires six steps of instantiation in a target container.

We measure performance when a client requests from 1 to 30 instantiations in two different environments (sufficient and insufficient available nodes, respectively, in the local HUST-VO located at the Huazhong University of Science and Technology). By default, the HAND infrastructure in HUST-VO can adopt the available resources from another VO for CFD computing that's at the Shenzhen Institute of Advanced Technology as backup. To compare our provisioning solutions' efficiency, we specify the different provisioning levels manually in four experiments.

Figure 3 shows that as the number of allocated nodes increases, the speedups achieved by both node- and VO-level provisioning increase

greatly. In Figure 3a, the node-level speedup is higher than the VO-level one because forcedly using VO-level provisioning inside a VO requires an additional time cost for security and consistency maintenance. However, the VO-level speedup in a multiple-VO environment is much higher than the node-level one when crossing VOs (see Figure 3b). The highly asynchronous mechanism is more adaptable in heterogeneous environments. From the vertical axis, we find that the increase in the number of dependent services on leased resources has less effect on provisioning efficiency because provisioning agents execute the instantiations on a physical node sequentially. In addition, the improvements achieved via service-level provisioning are small because the higher levels will propagate the provisioning in parallel but will conduct the enhancement of service-level limits in the reloading time in separate containers. HAND adopts the most efficient approach for provisioning regardless of the environment, proving that it can achieve high self-maintainability.

Because no universally optimal solution for dynamic-instantiation of community services exists, the best approach is the one that most closely matches VO users' provisioning demands. We compare the four provisioning levels in terms of scale, dependency-awareness, consistency-awareness, automation, efficiency, and availability.

In our current work, we're developing our HAND system further in two primary directions. First, we're expanding the provisioning targets from Web services and legacy software to operating systems, virtual machines, and even

Table 1. Provisioning granularities for distributed environments.

	Container-level	Service-level	Node-level	Virtual organization-level
Scale	Small	Small	Big	Very big
Dependency aware	N/A	Weak internal service dependency	Strong service dependency	Very strong VO dependency
Consistency aware	N/A	N/A	Simple	Complicated
Automation	Low script	Low script	High script, workflow	High distributed workflow
Efficiency	Lowest	Low	High inside the VO	High among the VOs
Availability	Low	High; affected by internal dependencies	High; affected by dependencies	High; affected by dependencies, consistency, and delegation

networks. Second, we intend to set up a neural network to automate provisioning according to IT infrastructures' current status. This will let us build a real self-configuring, self-healing, and self-propagating grid system. □

#### Acknowledgments

This work was partially supported by the National Science Foundation, China, under grants 90412010, 60673174, 60603058, and the ChinaGrid project funded by the Ministry of Education, China. It was also supported in part by the Mathematical, Information, and Computational Sciences Division subprogram at the Office of Advanced Scientific Computing Research, Office of Science, US Department of Energy, under contract DE-AC02-06CH11357, and by the US National Science Foundation's Middleware Initiative program.

#### References

1. Global Grid Forum, "Open Grid Service Architecture, Version 1.0," 2002; [www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf](http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf).
2. M.N. Huhns and M.P. Singh, "Service-Oriented Computing: Key Concepts and Principles," *IEEE Internet Computing*, vol. 9, no.1, 2005, pp. 75–81.
3. I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," *J. Computer Science and Technology*, vol. 21, no. 4, 2006, pp. 513–520.
4. J. Weissman, S. Kim, and D. England. "A Framework for Dynamic Service Adaptation in the Grid: Next Generation Software Program Progress Report," *Proc. 19th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS 05)*, IEEE CS Press, 2005, pp. 5–9.
5. L. Qi et al., "HAND: Highly Available Dynamic Deployment Infrastructure for Globus Toolkit 4," *Proc. 15th Euromicro Int'l Conf. Parallel, Distributed and Network-Based Processing (PDP 07)*, IEEE CS Press, 2007, pp. 155–162.
6. I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Proc. European Conf. Parallel Computing 2001 (Europar 01)*, Springer-Verlag, 2001, pp. 1–4.

7. H. Jin, "ChinaGrid: Making Grid Computing a Reality," *Proc. Digital Libraries: Int'l Collaboration and Cross-Fertilization (ICADL 04)*, Springer-Verlag, 2004, pp. 13–24.
8. V. Talwar et al., "Approaches for Service Deployment," *IEEE Internet Computing*, vol. 9, no. 2, 2005, pp. 70–80.
9. N. Sangal et al., "Using Dependency Models to Manage Complex Software Architecture," *Proc. Int'l Conf. Object Oriented Programming, Systems, Languages and Applications (OOPSLA 05)*, IEEE CS Press, 2005, pp. 167–176.

**Li Qi** is a PhD candidate in computer science at the Huazhong University of Science and Technology, China. His research interests are in distributed computing, dynamic maintenance, and services computing. Qi has an MS in computer science from Huazhong University of Science and Technology. Contact him at [quick.qi@gmail.com](mailto:quick.qi@gmail.com).

**Hai Jin** is a professor and dean of the School of Computer Science and Technology at the Huazhong University of Science and Technology, China. His research interests include computer architecture, virtualization, cluster computing, grid computing, network storage, and network security. Jin has a PhD in computer science from Huazhong University of Science and Technology. Contact him at [hjin@hust.edu.cn](mailto:hjin@hust.edu.cn).

**Ian Foster** is director of the Computation Institute at the University of Chicago and Argonne National Laboratory and the Arthur Holly Compton Distinguished Service Professor of computer science at the University of Chicago. His research interests include distributed computing, parallel computing, and computational science. Foster has a PhD in computer science from Imperial College, London. Contact him at [foster@mcs.anl.gov](mailto:foster@mcs.anl.gov).

**Jarek Gawor** was a senior software developer working on the Globus Toolkit at Argonne National Laboratory, where he was a codesigner and a lead developer of the Java WS Core component of the Globus Toolkit 4. Jarek has an MS in computer science from the Illinois Institute of Technology. Contact him at [gawor@mcs.anl.gov](mailto:gawor@mcs.anl.gov).