ROST: Remote and hot service deployment with trustworthiness in CROWN Grid

Jinpeng Huai\textsuperscript{a}, Hailong Sun\textsuperscript{a}, Chunming Hu\textsuperscript{a}, Yanmin Zhu\textsuperscript{b}, Yunhao Liu\textsuperscript{b,}\textsuperscript{*}, Jianxin Li\textsuperscript{a}

\textsuperscript{a} School of Computer Science and Engineering, Beihang University, Beijing, China
\textsuperscript{b} Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong

Received 29 March 2006; received in revised form 18 January 2007; accepted 19 January 2007
Available online 2 February 2007

Abstract

The main goal of our key project, the CROWN Grid, is to empower in-depth integration of resources and the cooperation of researchers nationwide and worldwide. CROWN exploits a service-oriented architecture based on OGSA. In CROWN, remote service deployment is highly desirable. To the best of our knowledge, however, there is no successful solution to ensure the enabling remote and hot service deployment in grid systems. Traditionally, remote deployment is supported in a cold fashion, which results in many disadvantages, such as low efficiency. Moreover, since the deployer and the target container may be in different domains, great security challenges arise when a service is deployed to a remote container. In this paper, we present ROST, an original scheme of Remote and hot Service deployment with Trustworthiness. By dynamically updating runtime environment configurations, ROST avoids restarting the runtime system during deployment. In addition, we include trust negotiation in ROST, which greatly increases the flexibility and security of the CROWN Grid. ROST has been successfully implemented. We conduct comprehensive experiments with real applications, and the results show that ROST is viable and significantly improves the service efficiency and quality of CROWN. We believe that the wide deployment of ROST would also benefit other grid systems.

Keywords: Distributed computing; CROWN Grid; Remote and hot deployment; Trust negotiation agent (TNA); Prototype implementation

1. Introduction

Grid computing promises to enable coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations [17]. As grids are very heterogeneous environments, there exist various resources including networks, computers, storage and devices, and the like, to better tackle the heterogeneity problem. Service-oriented grid architecture has recently been introduced, which is widely considered to be the future of grid computing [19]. Built on web services, OGSA [18] is the de facto standard for building service grids, in which various resources are encapsulated as services with uniform user interfaces.

The main goal of our key project, CROWN (China R&D Environment Over Wide-area Network) Grid, is to empower the in-depth integration of resources and cooperation of researchers nationwide and worldwide. The CROWN project was launched in late 2003. As illustrated in Fig. 1, a number of universities and institutes, such as Tsinghua University, Peking University, Chinese Academy of Sciences, and Beihang University, have joined CROWN, each contributing 50–100 computing nodes. More universities and institutes are invited to join the CROWN Grid. CROWN will also be connected to some world-famous grids, such as GLORIAD [7] and PRAGMA [8].

In past years, many key issues in grid computing have been extensively studied, such as information services, resource scheduling, and security issues. As an important issue, remote and hot service deployment, however, has not been fully addressed. Before a service is ready for invocation, it must be deployed in a service container, which provides a runtime environment. A grid is a highly distributed environment, in which numerous domains could be involved. The domains are usually geographically dispersed. It is highly desirable for a contributor to deploy its services within remote service
containers for multiple purposes. For example, in the CROWN Grid for applications in bioinformatics, there are many computing intensive applications such as BLAST. A computing node could easily become overloaded when multiple jobs arrive in a short time period. The heavy load can be balanced if the node is able to deploy one or more BLAST service replicas to remote nodes and then redirect some jobs. Similar requirements also exist in many other grid applications.

Traditionally, remote service deployment is supported in a cold fashion, which means that to deploy a new service, the runtime environment needs to be restarted. This results in many disadvantages because previously running services must be stopped, and they may have to resume or even restart their jobs, causing significant overheads. Therefore, hot service deployment, which does not need to restart the runtime environment while deploying services, has become increasingly important. With the availability of remote and hot service deployment, many applications will benefit, such as load balancing, job migration and so on.

Service deployment is actually not a new issue. Similar demands also exist in mobile agents [11] and active networks [13]. To the best of our knowledge, however, there is no successful solution to enable remote and hot service deployment in grid systems. The most updated Globus Toolkit (version 4) [4], the de facto standard for grid middleware, does not yet provide the function of remote and hot service deployment. This may be due to the great security challenges arising when a user deploys a service to a remote container. Here, we call a node a deployer, which intends to deploy a service, and the remote service runtime environment the target container, which is responsible for running and managing services being deployed. Without proper security mechanisms, a service provided by a deployer may be malicious, and the target container may be rogue or fragile. Also, the security policies of the deployer and the container could be incompatible. In an open grid environment, we cannot expect any deployer and its corresponding target container to set up a required trust relationship in advance. Moreover, it is too costly to always have to build trust across domains based on the traditional PKI infrastructure during remote deployment.

In this paper, we present our original work, ROST (Remote and hot Service deployment with Trustworthiness), which achieves its goal by dynamically updating the runtime environment configurations. ROST avoids restarting runtime systems during remote deployment. Moreover, we include trust negotiation in the ROST scheme, which greatly increases the flexibility and security of CROWN. The major contributions of this work are as follows:

- We identify the necessity of remote and hot service deployment in service grids, and their challenges.
- We propose an effective approach, ROST, to enable remote and hot service deployment. Also, we add trust negotiation into the scheme to meet general security requirements for grid environments.
- We implement the ROST scheme in the CROWN Grid with real applications, and evaluate the performance of ROST by means of comprehensive experiments.

The rest of this paper is organized as follows. In Section 2, we present our proposed approach to ROST in detail. The implementation experiences are introduced in Section 3. We present our experimental methodology and performance evaluation of ROST in Section 4. We discuss related work in Section 5 and conclude this work in Section 6.

2. ROST design

CROWN consists of numerous organizations, with each of them forming a domain, as illustrated by Fig. 2. The
public Internet usually connects domains. CROWN, as a service-oriented grid, encapsulates various resources, such as web services. In CROWN, a computer must be installed with a core component of CROWN middleware, the Node Server (NS), before joining the grid system. The computer installed with the Node Server middleware is also referred to as an NS. An NS serves as the service container, which provides a runtime environment for various services. Each NS usually belongs to a security domain. Every domain has at least one RLDS (Resource Locating and Description Service) to provide information services. The RLDS maintains the dynamic information of available services. And RLDS services from different domains can communicate with each other through SOAP messages, which allows users to discover resources across domains transparently.

Remote service deployment is needed when a deployer needs to deploy a service on an NS in a different domain. In this paper, we refer to deploying a service to an NS and deploying a service to a container interchangeably, which have the same meaning. A service is basically an entity that consists of an executable program, a description file, and several configuration files. Before a deployer’s services can be ready for invocation on a remote NS, two key issues must be addressed. The first is security, namely, to guarantee that the service provided by the deployer is not malicious and the environment provided by the remote container is safe to the service. The second is how to enable the service’s availability without restarting the remote container.

### 2.1. Overview of ROST

The basic procedure of ROST is as follows:

**Step 1:** the deployer sends a deployment request to a remote NS;

**Step 2:** the remote NS checks locally whether it can afford to run the new service; if yes, goes to Step 3;

**Step 3:** the remote NS checks whether the deployer has been trusted according to the local domain controller or its history information. If yes, it sends a trusted notification; otherwise, it initiates trust negotiation;

**Step 4:** the deployer checks whether the remote NS has been trusted. If yes, it sends a trusted notification, and goes to Step 5; otherwise, it initiates trust negotiation;

**Step 5:** if the negotiation successfully sets up the desired trust, the deployer initiates service deployment by transferring the service to the remote NS;

**Step 6:** the remote NS performs the hot deployment of the service;

**Step 7:** the remote NS acknowledges the success of the deployment.

### 2.2. Trust negotiation

CROWN is a large-scale decentralized environment, which makes it difficult, if not impossible, to build a pairwise trust relationship for all deployers and target containers in advance. The dynamic nature of a grid environment also implies that trust between a deployer and a target container be set up on demand at the time needed. In a remote service deployment scenario, however, we have to set up trust before a service is deployed.

#### 2.2.1. Selection of trust negotiation mechanisms

Several security infrastructures have been proposed for grid computing. For instance, in the Grid Security Infrastructure (GSI) [16], every user or computer is uniquely identified by an X.509 certificate, which is issued by a Certificate Authority (CA). A CA itself also has a certificate, which is further issued by another CA. When two individuals attempt to verify each other, they check the validation of the certificates by validating the chain of CAs. This fashion provides very limited capabilities of security control, and it is rarely possible to deploy such a global hierarchy of CAs in an open environment such as CROWN.

The other limitation of traditional approaches, such as GSI, is that they are designed for a closed security domain, where members can eventually determine the identity of every node. As CROWN spans multiple security domains, however, it is extremely difficult to determine another’s identity. To address security issues in an open network, attribute-based trust management mechanisms, such as RT (Role Based Trust Management) [26] and SPKI/SDSI 2.0 [14], have been proposed. These are based on a unified approach to specify and interpret security polices, credentials, and relationships, and they allow direct authorization for security-critical actions [12]. However, the attributes in credentials are often sensitive, which
may expose some private information of participants, such as a social security number. The above mechanisms do not deal with how to protect sensitive information.

ATN (Automated Trust Negotiation) [25, 32–35] is a new approach to access control in an open environment, which, in particular, successfully protects sensitive information while negotiating a trust relationship. With ATN, any individual can be fully autonomous. Two individuals can try to set up a trust relationship by exchanging credentials according to their respective policies. Compared to traditional mechanisms, ATN has the following advantages [35]: (1) Trust between two individuals that are not in the same security domain can be automatically established based on attributes of individuals in an open network; (2) One can define one’s own policies for accessing its own sensitive information; (3) Trust can be established directly without involving trusted third parties, other than credential issuers.

Based on the above observations, we solve the trustworthiness problem in ROST by adding a Trust Negotiation Agent (TNA) in the CROWN middleware, which is generally based on ATN protocol.

### 2.2.2. Negotiation scheme of ROST

In the ROST scheme, before a target container can trust a deployer, it requires the deployer to show its credentials and specify some critical attributes. On one hand, as pointed out before, credentials may be sensitive, so policies must be defined to protect them. The policies specify what conditions must be satisfied before the credentials can be exposed to the other side. On the other hand, to trust the container, the deployer may also require the container to show some credentials. Thus, the trust negotiation is mainly comprised of the exchange of credentials according to each partner’s respective policies.

Fig. 3 depicts the basic negotiation procedure in ROST between the TNAs of a deployer and a container. A trust negotiation has the advantage that trust can be established through cautious, interactive, and bilateral disclosure of credentials and access control policies while their autonomy and privacy are also maintained.

To make it clear, let us look at the simple example shown in Fig. 4. Suppose node D needs to deploy a service on target container T. D sends T a deployment request $R_{dep}$. T sets an access policy that only nodes with attribute-based credentials of both $C_{A1}$ and $C_{A2}$ are permitted to take the deployment operation. T responds with Policy $(C_{A1} \land C_{A2} \rightarrow R_{dep})$. D has such two credentials, but regards $C_{A2}$ as sensitive information. Thus, D sets a policy that containers with credential $C_{B1}$ are eligible to read $C_{A2}$. Then D replies with $C_{A2}$ and Policy $(C_{B1} \rightarrow C_{A2})$. Next, T presents $C_{B1}$ to D, and D sends $C_{A2}$ to T. The successful negotiation finally establishes the desired trust, and T’s deployment request is granted.

A negotiation process may be long and cause additional time delays. Also, in many cases, it is desirable for users to update a previously deployed service or to add/remove a service. It is a waste if the same negotiation is performed again. To improve negotiation efficiency, we design a TrustTicket for the ROST scheme. After a successful trust negotiation, a deployer can apply for a TrustTicket from the container. The container could issue a TrustTicket, which stores some critical security information. With this TrustTicket, the deployer need not negotiate again with the same container. Instead, it shows the TrustTicket. For the sake of better security, TrustTickets are signed by the containers and have a limited lifetime. In addition, to guarantee the termination of a negotiation, we define a timeout threshold for long negotiations.
2.3. Hot deployment

After a negotiation successfully sets up the desired trust, the container receives the service from the deployer and begins to deploy it.

The executable program of a service can be implemented with a wide range of technologies, such as JAVA, .NET, CORBA, and even scripts. A service has a description file, i.e. a WSDL file, which specifies how to invoke the service. In addition, some configuration files, such as WSDD and JNDI files, must be provided together with service, which are necessary for the container to be able to configure the service. Traditionally, for example, in the Globus Toolkit, the container loads the executable programs and performs the configurations statically, i.e., only once when the container is started. This leads to the disadvantage where the container fails to deploy new services. Although we can place a new service on this NS, the container fails to be aware of the arrival of the new service. On one hand, the executable program of the service is not loaded into the runtime environment; and on the other hand, the container does not include the configurations of the service. Therefore, to deploy a new service, the container has to be restarted, which is termed by us as cold deployment.

We achieve hot deployment by adding the RHD (Remote Hot Deployment) function block into the CROWN Node Server. A service is usually compressed into a single file, such as a GAR (Grid Archive) [4], which includes all necessary files of the service. In ROST, a compressed file is conveyed over the network using some communication protocol, such as FTP and SOAP, for which we have more words in Section 4. The RHD receives the compressed file and recovers the service contents. Next, the RHD has the executable program of the service loaded into the runtime environment. Also, the RHD actively updates the configuration of the container by pushing the configurations of the service into the container. The consequence is that when the service is requested, the container is able to locate the executable program, and invoke it by passing proper parameters, and return results properly.

We achieve hot deployment by adding the RHD (Remote Hot Deployment) function block into the CROWN Node Server. A service is usually compressed into a single file, such as a GAR (Grid Archive) [4], which includes all necessary files of the service. In ROST, a compressed file is conveyed over the network using some communication protocol, such as FTP and SOAP, for which we have more words in Section 4. The RHD receives the compressed file and recovers the service contents. Next, the RHD has the executable program of the service loaded into the runtime environment. Also, the RHD actively updates the configuration of the container by pushing the configurations of the service into the container. The consequence is that when the service is requested, the container is able to locate the executable program, and invoke it by passing proper parameters, and return results properly.

We achieve hot deployment by adding the RHD (Remote Hot Deployment) function block into the CROWN Node Server. A service is usually compressed into a single file, such as a GAR (Grid Archive) [4], which includes all necessary files of the service. In ROST, a compressed file is conveyed over the network using some communication protocol, such as FTP and SOAP, for which we have more words in Section 4. The RHD receives the compressed file and recovers the service contents. Next, the RHD has the executable program of the service loaded into the runtime environment. Also, the RHD actively updates the configuration of the container by pushing the configurations of the service into the container. The consequence is that when the service is requested, the container is able to locate the executable program, and invoke it by passing proper parameters, and return results properly.

3. Experiences with ROST implementation

In CROWN, services follow the WSRF specifications [1]. A complete service consists of several files, with each specifying some properties of the service.

- Executable programs. Such as Java classes, scripts, EJBs, and the like.
- One or multiple WSDL files. Description of interfaces and access protocols of a service.
- A WSDD file. Web Service Description Descriptor, description of service configuration for the service container.
- A JNDI configuration file. Description of WSRF resources of a service.
- A security configuration file. A description of the authorization approach and other security related information.

To facilitate the transportation and protection of services, we compress a service into one single file. We have adopted the Globus Toolkit’s GAR. In addition, we have extended GAR so that it is able to contain multiple types of executable programs and description files.

3.1. ROST architecture

As shown in Fig. 5, ROST is composed of several components, while our discussions focus on the two major ones, i.e., TNA and RHD. TNA is responsible for trust establishment between a pair of deployers and containers, and RHD is for remote and hot service deployment. The SCC (Service Container Configuration) is the abstract of various configurations of service containers. Indeed, each deployment operation results in an update to SCC.

3.2. TNA

As illustrated by Fig. 6, TNA has mainly four components as follows.

- TrustTicket manager: TNA issues new TrustTickets for requesters or validates TrustTickets based on the local Ticket Repository.
- Strategy enforcer: The negotiation strategy [36] is used to determine when and how to disclose local credentials and policies. The Strategy Enforcer makes decisions about the negotiation states, such as success, failure, or continuance.
Fig. 6. TNA.

- Compliance checker: Determines which local credentials satisfy the requestor’s policies or whether the requestor’s credentials satisfy local policies.
- Credential chain construction: For trust negotiation in open networks, an access control decision often involves finding and constructing a credential chain that delegates authority from the source to the requester, when the credentials are not stored locally. The main function of this component is to discover and collect necessary credentials.

In the ROST scheme, TNA is deployed on both deployers and target containers. The procedure of trust establishment is illustrated in Fig. 6. If a requestor has a valid TrustTicket, then the TNA calls the TrustTicket Manager to make an access decision. Otherwise, a trust negotiation is triggered. When the requestor discloses its policies, the Strategy Enforcer decides whether the negotiation should continue. If so, the TNA calls the Compliance Checker to make a corresponding verification to ensure which credentials should be provided, and then it responds with the necessary credentials and policies. In some cases, if the credentials are not available in a local Credential/Policy Repository, the Credential Chain Construction is called to dynamically retrieve the necessary credentials. Similarly, when the requestor submits its credentials, the TNA calls the Compliance Checker to make a corresponding verification to ensure whether the credentials satisfy local policies and make access decisions. To speed up the negotiation process, if the TNA estimates that it may discover the same credential chain for different negotiation processes, it caches the chains to avoid frequent retrieval.

In TNA, we adopt a refined RTML (Role-based Trust Management Language Markup Language) to represent both access control policy and attribute-based credentials. When credential storage is distributed, the goal-directed algorithm [25] ensures that all credentials available can be discovered and collected. In the ROST design, the TrustTicket has the form of (subject, issuer, issueDate, expirationDate, signature). It is an identity assertion represented by XML with a short lifetime assigned by an issuer.

In addition, negotiation information exchange between participants must rely on a secure communication protocol such as SSL/TLS to prevent eavesdropping, man-in-the-middle attacks, replay attacks, and so on. In ROST, we conform to WS-Security and WS-Conversation specifications for SOAP message protection.

3.3. RHD

RHD enables remote hot deployment as well as providing a convenient way for local hot deployment. RemoteDeployment and AutoDeployment, as shown in Fig. 5, are responsible for remote and hot service deployment, respectively.

3.3.1. RHD APIs and supporting tools

We design APIs for both remote and local deployment, through which users are able to develop a friendly UI. There are basically three types of deployment operations: deploy, update, and undeploy. We define nine APIs to support these deployment operations as follows.

1. deploy (String garFilePath)
2. deployByFTP (URL garFileURL, String user, String password)
3. deployBySOAPAttach (String garFilePath)
4. update (String garFilePath)
5. updateByFTP (URL garFileURL, String user, String password)
6. updateBySOAPAttach (String garFilePath)
7. undeploy (String garFileName)
8. undeploy (String serviceName)
9. getAllDeployedServices().

Note that (1)–(3) are interfaces for deploying a service. The first one (1) is for deploying a service locally; while (2) and (3) provide different interfaces for remote deployment. For updating deprecated services, (4)–(6) define three interfaces. For removing services from service containers, (7) and (8) define two interfaces. We define (9) for querying all services deployed in a service container as follows.

In ROST, we take advantage of existing tools as well as developing our own tools to fulfill the processing tasks of service deployment. We have designed a GARUnzipper for uncompressing GAR files to recover all contents of a service. Existing tools, such as WSDL4J from IBM, parsers of WSDD, JNDI, and security configuration files, are used to parse the description and configuration files of a service. Finally, we make use of Apache ANT to coordinate these tools to deal with various deployment operations.

3.3.2. Remote deployment

After mutual trust is successfully established, ServiceReceiver is called on to receive the GAR file and uncompress it using GARUnzipper. Then the underlying deployment functions are called on to perform the corresponding operations.

A service container must include various configurations of the deployed services. Indeed, the key to hot deployment is to dynamically update the configuration of SCC. Relevant configurations include executable programs, WSDL description, and WSDD, and JNDI configurations. For example, when a new service implemented with JAVA needs to be deployed, we have to let SCC load the JAVA classes of that service.
Updating or un-deploying an existing service should be handled carefully, since other services or users might be using it. Simply updating or undeploying a service without adding special measures may lead to unexpected service interruptions to users. To solve this problem, we add a reference counter for each deployed service. The initial value of a counter is zero, and the value increases or decreases by one each time the service is invoked or completed. When an update or undeployment request arrives, we first check the counter of the service. A service is ready to be updated or undeployed only if the reference counter is equal to zero.

3.3.3. Auto deployment

Besides remote hot deployment, the RHD component also provides a convenient method to hot-deploy services to local containers.

A file folder is specified to receive GAR files and an EventListener keeps listening to the events associated with that folder. The EventListener is interested in two types of events: the arrival of new files, and the deletion of existing files.

Suppose an event e caught by EventListener is passed to the EventAnalyzer for analysis and further processing. Based on the contents of an event, the EventAnalyzer will call different underlying deployment functions. In Fig. 7, we provide the pseudo code for this process.

As a result, users may deploy/update/undeploy a local service by simply storing/replacing/removing its GAR file to/in/from a folder. They need not care about the underlying processes, and services are deployed/undeployed automatically and transparently.

3.4. Real applications

In CROWN, the ROST scheme is currently employed by two real applications: Huge Integer Factorization (HIF) and the University Digital Museum (UDMGrid).

In HIF, we adopt the SIQS (Self Initializing Quadratic Sieve) [29] algorithm. With ROST, as illustrated in Fig. 8, the resource broker can dynamically deploy multiple siever services to available NSs. As a result, the time needed for factorizing a huge integer can be significantly shortened.

In UDMGrid, a WEB portal is provided for users to invoke FavoriteSeek services from multiple university digital museums. When too many users try to invoke the FavoriteSeek service in an NS, it may deploy the service to other available NSs so as to balance the heavy load. Due to page limitations, we do not discuss the two applications further, and details can be found in [24].

4. Performance evaluation

The ROST scheme is implemented as a core component of CROWN middleware. We evaluate the performance of ROST by comprehensive experiments based on the abovementioned real applications.

4.1. Experimental environment

The experiments were conducted across two domains connected by the Internet. The deployer resides in Tsinghua University (THU in Fig. 1), while the target NSs (i.e., target containers) are located in Beihang University (BUAA in Fig. 1). The deployer has a Pentium III 1.6 GHz CPU and 512M memory, with a 10 Mbps connection to the Internet. Remote NSs reside in a 32-node cluster, with each of the NSs having two Intel Xeon 2.8 GHz CPUs and a 2G memory. The cluster is connected to the Internet through a 100 Mbps connection. No other tasks are running on any node except the necessary CROWN middleware.

4.2. Performance metrics

We use the following metrics to evaluate ROST.

Deployment response time. It is important that a remote service deployment introduces a shorter response time. When multiple concurrent deployment requests are sent to a single NS, the deployment response time increases.
Task execution time. A task here means a collection of independent jobs, while a job means an invocation of a specific service. Given a task, we are concerned with its total execution time.

Job processing efficiency. Given a job, we define its length, $t_0$, as the time needed for a clear NS to execute the job. In experiments, each job has a real execution time, $t_r$, on NSs. We define the processing efficiency $\delta$ of a job by $\delta = \frac{t_0}{t_r}$. Mostly, $\delta$ is less than one and greater than zero. We carefully select 30 different real applications and implement them into CROWN. We run each job 100 times on a clear NS, and measure its running time, taking the average as the job length.

4.3. Experimental results and analysis

We execute each experiment 100 times and report the average.

4.3.1. Deployment response time

In our first experiment, we evaluate the performance of ROST in terms of the deployment response time. The deployer in Tsinghua University issued concurrent deployment requests to a node server in Beihang University. We vary the ways of service GAR file transfer, FTP, and SOAP attachment. Each GAR file is of size about 6K bytes. Fig. 9 shows the average deployment response time as a function of the number of concurrent requests. When there is only one request each time, the response time of ROST is as short as seven seconds. In contrast, the cold deployment needs as long as 30 s to merely stop and restart the service container so as to load a new service. With an increasing number of concurrent requests, the average response time increases roughly linearly.

When the number of concurrent requests reaches 30, the average response time is about 52 s. We also observe that SOAP transfer has a similar performance to the FTP mechanism.

4.3.2. Task execution time

We then study how well ROST can help to achieve load balancing. In the second experiment, two schemes are compared, with and without ROST. There are 20 NS’s available for processing jobs, while initially only a fraction of the nodes are deployed with the required service.

Fig. 10 shows the task execution time with a different number of nodes initially deployed with the service. With ROST, the task execution time is significantly reduced, as a node may easily deploy its service to other relatively idle nodes. In some specific cases, the maximum improvement can be four times as fast. When the fraction of nodes initially deployed with the service increases, the effect of time reduction becomes less.

4.3.3. Job processing efficiency

The third experiment was designed to evaluate how ROST helps to improve the job processing efficiency. Here, we still compare the system performance with and without ROST. Without ROST, if the node is executing another job when a new job request arrives, the node first completes the current job, and then restarts the service container so as to deploy the new service in a cold fashion. In contrast, with ROST, the node deploys the new service without restarting the service container, even if the node is executing other jobs at the same time.

Figs. 11–13 show the distributions of the job processing efficiency of 10, 20, and 30 jobs, respectively. In these figures, the $x$ axis represents the job processing efficiency, and the $y$ axis represents the percentage of nodes whose efficiency is greater than the one specified in the $x$ axis. We can see that with ROST, more than 80% of jobs have a job processing efficiency of nearly 90%. Without ROST, 80% of job efficiency is as low as 35%.

Fig. 14 shows the overall job completion time as a function of a different number of jobs. As we can see, with ROST, the overall job completion time is reduced significantly. When there are 30 jobs, the reduction of time is as much as 36.8%.

5. Related work

The Globus Toolkit is the most famous grid middleware, and it has begun to support service-oriented grid computing based on OGSA since version 3. However, even in the updated
release, version 4, remote and hot service deployment is not supported. A grid service is actually built on the Web service, and extended to include functions such as state and life cycle management. For Web services, a number of middlewares, such as Apache Axis [5], JBOSS [15], and Microsoft .NET [6], have partly implemented dynamic service deployment, i.e., deploying a local service without restarting service containers. However, a Web service is much simpler than a grid service, e.g., web services are normally stateless, and so web service middlewares cannot apply to grid environments. Also, most of them only consider local deployment.

Friesen et al. [21] proposed a method for hot service deployment in an ad hoc grid environment based on OGSI, which is now replaced by WSRF. To ensure security, they make use of the sandbox, which can restrict the service functions. DistAnt [23] extends the Apache Ant built file environment to provide a flexible procedural deployment description, and provides a solution to remote and hot service deployment based on Globus Toolkit 3. It does not provide any security mechanism for remote deployment. Baude et al. [10] proposed a solution for the deployment and monitoring of applications written using ProActive, which is a Java-based library for concurrent, distributed, and mobile computing. It does not consider undeployment and updating active objects.

To provide mutual trust and to ensure the safe execution of remote programs, much effort has been made. For example, ActiveX Controls [2] and Java Applet [3] have designed security mechanisms for the safe execution of mobile codes, and [9,22,27,28,31] were proposed to deal with the security of mobile codes. Rubin and Geer [30] provide typical solutions for mobile code security, such as sandbox and signed code. These solutions cannot satisfy the security requirements of CROWN, where we do not want to restrict any execution of services, and a digital signature alone cannot solve the trust problem.

DisCo [20] middleware infrastructure facilitates the construction and deployment of decomposable services in environments without a stable trust relationship across multiple security domains. It does not deal with how to protect sensitive credential information and access control policies.

Our proposed approach, ROST, is built on CROWN Grid’s real need for remote hot deployment of grid services. Large-scale and loosely connected grids are targeted. Any
grid middleware is able to support remote and hot service deployment with trustworthiness by implementing the two ROST components, TNA and RHD.

6. Conclusions and future work

Our key project, CROWN Grid, strives to integrate valuable nationwide and worldwide Internet resources. In CROWN, remote and hot service deployment is in high demand. In this paper, we present our remote and hot service deployment with a trustworthiness (i.e. our ROST) scheme. With ROST, a service can be deployed to a remote container in a different security domain in a hot fashion, which significantly improves service efficiency and quality. We implement ROST in CROWN, and the experiments based on real applications demonstrate the effectiveness of ROST. We believe that the wide deployment of ROST will benefit many grid systems.

Future work will go three directions. First, the current trust negotiation of ROST causes much time overhead. We will explore more relevant trust mechanisms, and further improve the trust negotiations' efficiency. Second, CROWN is rapidly growing, and a large number of computing sites will be included. More sophisticated experiments will be performed to guarantee that ROST can be adapted to large-scale networks. Third, we are developing an integrated grid middleware including job scheduling, information service, and security, and others. We are still striving to integrate ROST well with other components to provide better support for grid systems.

Acknowledgements

This work was supported by the National Natural Science Foundation of China under Grant No. 90412011 and 60473010, the National Natural Science Funds for Distinguished Young Scholars under Grant No. 60525209, and the National Grant for Fundamental Research 973 Program of China under Grant No. 2005CB321803.

References

Jinpeng Huai is a Professor and Vice-President of Beihang University. He serves on the Steering Committee for Advanced Computing Technology Subjects, the National High-Tech Program (863) as Chief Scientist. He is a member of the Consulting Committee of the Central Government’s Information Office, and Chairman of the Expert Committee in both the National e-Government Engineering Taskforce and the National e-Government Standard office. Dr. Huai and his colleagues are leading the key projects in e-Science of the National Science Foundation of China (NSFC) and Sino-UK. He has authored over 100 papers. His research interests include middleware, peer-to-peer (P2P), grid computing, trustworthiness, and security.

Hailong Sun is a Ph.D. candidate in the School of Computer Science and Engineering, Beihang University, Beijing, China. He received his B.S. degree in Computer Science from Northern Jiaotong University in 2001. His research interests include grid computing, web services, peer-to-peer computing, and distributed systems.

Chunming Hu is a research staffer in the Institute of Advanced Computing Technology at the School of Computer Science and Engineering, Beihang University, Beijing, China. He received his B.E. and M.E. in the Department of Computer Science and Engineering in Beihang University. He received the Ph.D. degree in School of Computer Science and Engineering of Beihang University, Beijing, China, in 2005. His research interests include Peer-to-Peer and Grid Computing, distributed systems, and software architectures.

Yanmin Zhu is a Ph.D. candidate in the Department of Computer Science, Hong Kong University of Science and Technology. He received his B.S. degree in Computer Science from Xi’an Jiaotong University, Xi’an, China, in 2002. His research interests include grid computing, peer-to-peer networking, pervasive computing, and sensor networks. He is a member of the IEEE and the IEEE Computer Society.

Yunhao Liu received his B.S. degree in the Automation Department from Tsinghua University, China, in 1995, and an M.A. degree in Beijing Foreign Studies University, China, in 1997, and an M.S. and a Ph.D. degree in Computer Science and Engineering at Michigan State University in 2003 and 2004, respectively. He is now an assistant professor in the Department of Computer Science and Engineering at Hong Kong University of Science and Technology. His research interests include peer-to-peer computing, pervasive computing, distributed systems, network security, grid computing, and high-speed networking. He is a senior member of the IEEE Computer Society.

Jianxin Li is a Ph.D. candidate at the School of Computer Science in Beihang University. He is currently working on the CROWN Grid project funded by the National Science Foundation of China. His current research interests include information security, trust management, and grid computing.