CROWN Node Server: An Enhanced Grid Service Container Based on GT4 WSRF Core

Hailong Sun, Wantao Liu, Tianyu Wo, Chunming Hu
School of Computer Science, Beihang University, Beijing, China
{sunhl, liuwt, woty, hucm}@act.buaa.edu.cn

Abstract

WSRF core is an open-source implementation of grid service container in Globus Toolkit 4 (GT4). However, GT4 WSRF core is far from a full-fledged grid service container. Basing on the Globus work, we develop CROWN Node Server, an enhanced grid service container. Besides basic functions of a WSRF grid service container, CROWN Node Server encompasses features including remote & hot service deployment with trustworthiness, monitoring, logging and management, etc, which are of paramount importance to build applications using service grid technologies. CROWN Node Server has been successfully adopted and widely deployed in CROWN Grid environment to support a wide range of service grid applications. We conduct comprehensive experiments to evaluate the performance of Node Server, and the comparing results with GT4 WSRF core are presented as well.

1. Introduction

With the wide acceptance of OGSA (Open Grid Service Architecture) [8] architecture, Web service technologies have been successfully incorporated into grid computing to deal with resource heterogeneity and other important issues. And the resultant service grid is generally regarded as the future of grid computing [9]. Many specifications, including OGSA, OGSI, WSRF, WSDL, SOAP, etc, are introduced to standardize service grid technologies. In service grid, various resources such as computers, storage, software, and data are encapsulated as services (e.g. WSRF services). As a result, resources can be accessed through standard interfaces and protocols, which effectively hides the heterogeneity. To make resources able to be accessed in the form of services, a runtime environment is necessary to provide running support for services and their instances. The runtime environment is usually referred to as *service container*. For each resource that wants to participate in grid, there must be a service container to provide necessary support. Usually, a service container is installed at a grid node, and services that encapsulate resources at that node are deployed onto the service container.

Globus Toolkit 4 (GT4) [1] is a famous open-source implementation of service grid technologies. It provides a set of loosely-coupled components including services & clients, libraries, and development tools, and these components are used to build grid-based applications and services. Among these components, WSRF core, a grid service container, provides a common runtime for WSRF services. GT4 contains three implementations of WSRF core, which are Java, C and Python runtimes. However, all the three runtimes simply implement the WSRF specifications, using a light-weight runtime on the basis of Axis [2] engine to process user requests. The middleware produced by OMII [3] also provides a web service container powered by Axis. The service container from OMII is featured by its process-based access control mechanism called PBAC.

Our CROWN (China Research and Development Over Wide-area Network) [4] project aims to support large-scale resource sharing and coordinated problem solving by using service grid and other distributed computing technologies. CROWN project was started in late 2003. As illustrated in Figure 1, a number of universities and institutes, such as Tsinghua University, Peking University, Chinese Academy of Sciences, and Beihang University, have joined CROWN. By now, CROWN nodes are distributed across multiple world-wide sites, e.g. nodes from UK, Australia and Hong Kong. In future, CROWN will also be connected to some world-famous grid testbed, such as GLORIAD [5] and PRAGMA [6].

In CROWN development, we find many real requirements for service containers that can not be met by GT4 WSRF core implementation. Basing on GT4's

Java WSRF core, we do several important extensions to produce CROWN Node Server, a full-fledged grid service container. Our extensions include remote & hot service deployment, resource status monitoring, logging, management and dynamic performance statistics, etc.

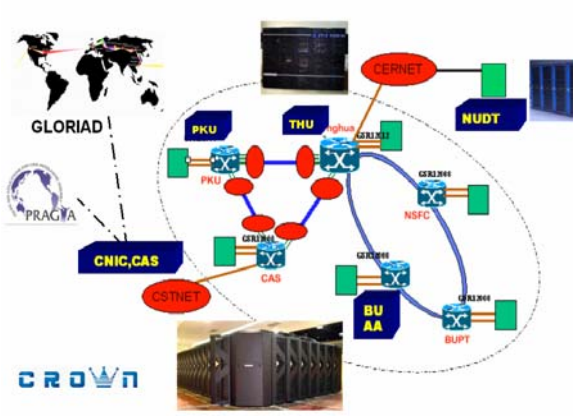

Figure 1. CROWN Grid

The rest of the paper is organized as follows. In section 2, we describe the system architecture and design issues of CROWN Node Server. Implementation details are presented in section 3. In section 4, we show the results of performance evaluation. And we conclude this paper in section 5.

2. System Architecture

CROWN Node Server is implemented on the basis of GT4 Java WSRF core, and Figure 2 shows its system architecture. GT4 provides a stable implementation of WSRF specification family and a light-weight embedded runtime. However, these basic functions are not enough to satisfy the requirements for service container in real grid environments.

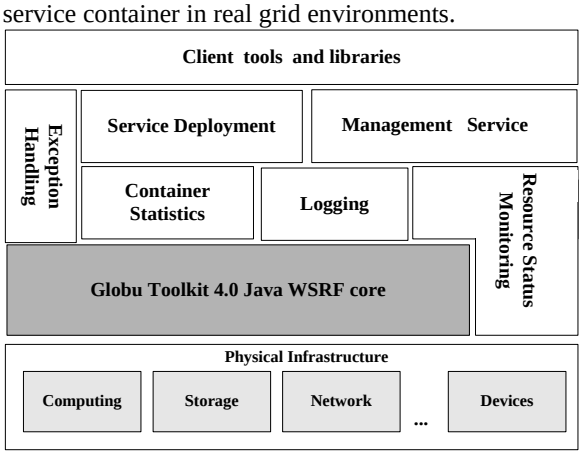

Figure 2. System Architecture of Node Server

First, as core elements in service grid, services should be deployed onto a service container before they can be invoked. In order to avoid interrupting the running of existing services, new services should be deployed in a hot manner, i.e., the service container does not have to be restarted. As grid is a highly distributed environment, deploying a service onto a remote container is highly desirable. Moreover, for the potential security risks, mechanisms should be provided to guarantee the security in service deployment. In Node Server, a base service is designed to address service deployment issues.

Second, Node Server has the functions of collecting and publishing status of underlying resources at a grid node. Resource status like CPU load and available memory, which changes dynamically, is critical to job scheduling. A resource monitoring service based on asynchronous notification is responsible for collecting and publishing resource status information dynamically.

Third, as a Node Server is the basic running environment for services, it should be configurable according to capacity of underlying resources and status of user requests so as to improve resource utilization and job processing efficiency. We develop a service, *container statistics*, to dynamically measure container performance data. Additionally, a container management service is designed to configure Node Server based on observed container statistics. Moreover the management service provides functions of disabling/enabling deployed services as well.

Fourth, GT4 provides log information with Apache Log4j tool [7], but this is not enough to log various messages in Node Server. Each instance of logger in Log4j is bound to a Java class, and all logging messages associated with a Java class including debug, warning, info, error and fatal messages are printed to the same destination. From this point of view, Log4J is an excellent and popular tool for software debugging. However, Log4J is not fit for logging application level information like events of service deployment and service invocation. The diversity of application level information can not be simply characterized by the four levels of Log4J. Therefore, we develop a logging service to fulfill the function of application level logging.

In addition, all possible exceptions are well defined so as to provide users with detail information in case of exceptional running situation. For developers, Node Server provides a set of command line tools and Java class libraries to access to the functions of base services.

As shown in Figure 2, most of the extensions are in the form of base services that run on top of GT4 Java

WSRF core. In practice, the main principle of Node Server design is to provide an enhanced grid service container with as few as possible modifications to GT4 code. Thus, our extensions are very flexible and convenient to deploy on a GT4 service container.

3. Implementation Experience

In this section, we present implementation details of various components in CROWN Node Server, including service deployment, resource monitoring, container statistics and management, logging, exception handling and client tools.

3.1. Service Deployment

With CROWN Node Server or Globus Toolkit 4, various grid resources are encapsulated into WSRF services. Before these services can be invoked, they need to be installed into Node Server and be properly configured. This process is usually referred to as *service deployment*. And the users or software agents that perform deployment operation are called *service deployers*.

First, the distributed nature of grid systems and applications brings forward the requirement for deploying services onto a remote container. Second, to maintain the availability of existing services, new services should be deployed in a hot manner. In other words, the deployment of new services should not force the service container to be restarted because restarting the service container will terminate the normal running of other services. Third, there should be a mechanism to dynamically establish the trust relationship between service deployers and service containers so as to ensure the security of deployment. A service deployer and the target service container can be located in different security domains. Thus potential security risks can be introduced by malicious service or rogue containers. For example, if a service that contains malicious code from an untrusted deployer is deployed onto a service container, the running of the malicious service can crash down the service container. In a word, the characteristics of grid environment require service to be deployed in a remote, hot and secure manner.

With Node Server, we propose and implement ROST (Remote and Hot Service Deployment with Trustworthiness) scheme to address this issue. In the following, we briefly describe the rationale of ROST, and more details are available in our previous work [10] [11].

Before being deployed, a service exists in the format of a GAR [1] file defined by Globus. The general procedure of deployment involves trust negotiation, obtaining GAR files, validation checking, configuring and updating the service container. The deployment function is implemented as a WSRF service. Both FTP (File Transfer Protocol) and SOAP with attachment are adopted to transfer GAR files to a remote Node Server so as to support remote service deployment. The validation checking step is to check whether the service configuration files including WSDD (Web Service Deployment Descriptor) and JNDI(Java Naming and Directory Interface) files conform with the defined schemas.

To implement hot service deployment, a daemon thread is designed to be running continuously for detecting the events of deployment and accordingly updating the configuration of Node Server. In GT4, a single Java class loader is used to load Java classes in all services' implementation. Such a design does not allow to undeploy a service without restarting the service container. We redesign GT4's Java class loader to make each service has a distinct class loader. The class loader of a Java thread dynamically loads Java classes related to the requested service when user requests arrive. When a service is undeployed, the corresponding Java class loader is also destroyed. The redesigned class loader mechanism ensures services are deployed/undeployed in a hot manner.

ATN (Automated Trust Negotiation) technology[12] is used to establish dynamic trust relationship between a deployer and the target service container. Two TNAs (Trust Negotiation Agent) are deployed respectively on the sides of a service deployer and the service container, and they exchange necessary credentials iteratively to establish trust relationship with ATN technology. If the trust relationship is established successfully between the deployer and container, the deployment process will start; otherwise, the deployment will fail. With ATN, service deployment only happens between trusted entities. Such a mechanism reduces security risks caused by malicious services and rogue service containers.

Besides deploying a service, Node Server also supports undeploy and redeploy functions that are used to remove and update deployed services respectively.

3.2. Resource Monitoring

Underlying resources form the physical infrastructure for components and applications in upper layers. The status of underlying resources has great impact on the whole grid systems in terms of functional and non-functional aspects. It is important

to design a service to monitor resource status and provide it for schedulers and other components. As resources are in the form of services in service grid, each node of a grid system has a service container installed. Therefore it is a straightforward way to deploy a resource monitoring service onto a service container so as to monitor the node resource status.

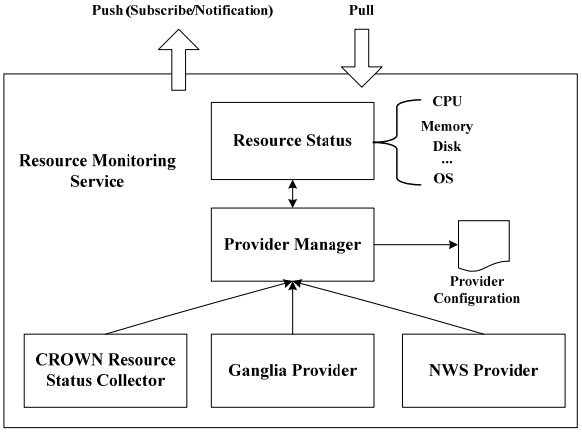

Figure 3. Resource Monitoring

The resource status monitoring function is also implemented as a WSRF service running in Node Server. Figure 3 presents the internal components of the resource monitoring service. The resource status we are concerned with includes both static and dynamic information. The static information involves OS type, file systems, IP address, host name, number of CPU, CPU frequency, total memory size, total disk size, etc; while the dynamic information includes available memory, CPU load and available disk space.

As there are many successful monitoring tools like Ganglia [13] and NWS (Network Weather Service) [14,15], the resource monitoring service in Node Server does not aim to reinvent the wheel. Instead, we focus on how to integrate monitoring results from those tools and how to make users retrieve monitoring information conveniently.

We design an extensible monitoring framework to allow other monitoring tools to be incorporated. For each monitoring tool, a corresponding provider is responsible for obtaining information from the monitoring tool and converting the information into the format defined by our monitoring service. With a configuration file, monitoring providers can be configured in terms of specific items that users are concerned with. A provider manager is designed to manage the providers that are configured, for example starting & stopping a provider and specifying intervals of information updating. A WSRF resource, *Resource Status*, is defined to represent resource status. Any updating of resource status information from a provider will cause *Resource Status* to be updated accordingly. Additionally, we develop CROWN resource status collector, a simple monitoring provider implemented with a Java thread, to serves as the default monitoring provider. With this provider, users can monitor resource status on Windows and most Linux platforms without installing other monitoring tools.

And the resource monitoring service provides users with two kinds of interfaces: pull and push. The push mechanism that is implemented using WSRF notification mechanism allows users to subscribe interested topics, and users will be notified when relevant resource status changes.

3.3. Container Statistics and Management

The service container provides the basic runtime environment for all services in grid systems. Hence the performance of application level services is largely dependent on the performance of the service container. In grid environment, both resource status and user requests are dynamically changing. As a result, the performance of the service container also keeps changing. Thus it is desirable to obtain the performance data of service container and tune its configuration dynamically so as to make it works in a good state. For example, if there are a great many incoming user requests and the available resource capacity of the grid node is powerful, we can increase the size of service thread pool and request queue to improve the throughput of the service container. To achieve this goal, we propose a container statistics service and a container management service.

The container statistics service is developed to measure the performance of Node Server dynamically. The performance metric we are concerned about includes average response time, throughput, success rate of user requests, status of the thread pool, status of request queue and continuous running time.

For the former three metrics, we insert a hander, ContainerStat handler, into the request chain of Axis engine and the response chain as well, as shown in Figure 4. With this handler, all the information about request and response is collected, and calculated to obtain real time performance. The latter three metrics are measured by code inserted into GT4 implementation. All the statistical data are serialized to XML files stored in disk, and can be obtained through a base service, ContainerStat service.

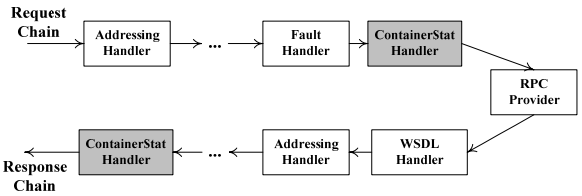

Figure 4. ContainerStat Handler

With the performance information obtained from ContainerStat service, system administrators can tune the configuration of Node Server to achieve specific goal, e.g. improving resource usage, via a container management service. For example, if the length of current request queue is very big and capacity of local node is powerful, the size of the thread pool can be increased to allow more requests to be processed simultaneously. We also develop a GUI management tool to provide administrators with viewing statistical data and performing corresponding configuration. Moreover, with the container management service, the services deployed onto a service container can be enabled/disabled.

3.4. Logging Service

During the process of running, Node Server generates many events, such as service invocation, service deployment/redeployment/undeployment and resource creation. These events are useful for auditing, accounting and failure analysis. In section 2, we analyze why logging tools like Apache Log4J can not fulfill this purpose. Hence, a logging service is proposed to record the log information and provides this information to users through WSRF service interfaces.

Figure 5 shows the structure of CROWN logging service. Inside of the logging service, a log queue manager (LQM) is responsible for managing a log queue. Various logging events flow into the log queue by accessing LQM, and LQM can be configured with what type of events are accepted. Two XML serializers, RollingFile and DailyRolling, are defined to serialize events in the log queue. The RollingFile serializer will write data to a new file when the size of an old file amounts to a specified value, while the DailyRolling serializer write data to a new file for each day. Users can query log data by time intervals and event types through a set of WSRF service interfaces.

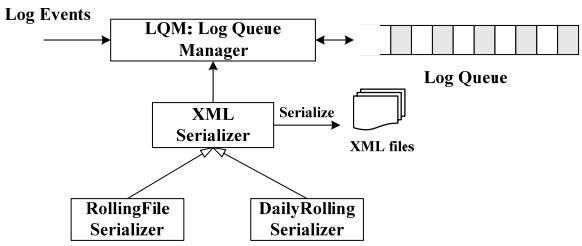

Figure 5. Logging Service

3.5. Exception Handling and Client Tools

Well-defined exceptions can provide much detailed information for users when unexpected situation occurs. For all possible exceptions that may occur in our extended services, we provide a set of well-defined Java classes to describe details of exceptions. When exceptions occur, users can quickly know what happens behind the exceptions and take corresponding measures.

To facilitate users to use our extended functions, we write a set of command line tools and Java libraries. With these tools and libraries, users do not need to write complex client code to access base services provided by CROWN Node Server.

4. Performance Evaluation

In this section, we present the performance evaluation of CROWN Node Server. As our work is based on extensions of GT4 (Version 4.0.0) Java WSRF core, we will show the performance comparison of Node Server and GT4 (Version 4.0.0) implementation. Moreover, we evaluate how each base service influences the total performance of Node Server.

4.1. Experimental Setup

The experiments are conducted on two nodes of a cluster. Between the two nodes, there is an Ethernet connection of 1000Mbps. Each node has dual Intel Xeon 2.8 GHz CPUs, 2 GB DDR Memory and Red Hat Linux OS. One node serves as the server node that has Node Server and GT4 (version 4.0.0) Java WSRF core installed, while another node is the client node that sends concurrent requests to the server node using Java thread technology.

We use the following metrics to do the performance evaluation.

(1) *Throughput*. This metric is used to evaluate how many requests can be processed in one second.

(2) *Average response time*. This metric reflects the processing efficiency under different scenarios.

(3) *Success rate*. The metric of success rate is defined as *sr = n/N*, where n is the number of requests processed successfully and N is the total number of requests.

All the experiments are repeated 10 times, and we report the average results.

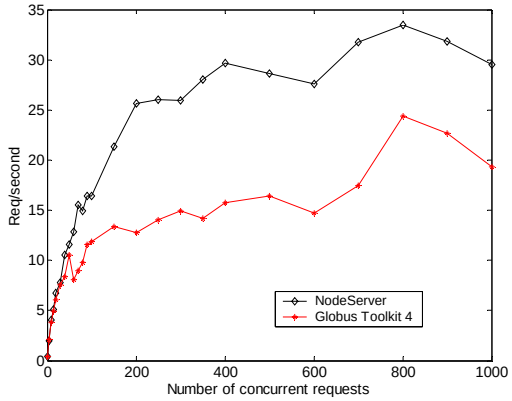

Figure 6. Throughput vs. Num of concurrent requests (echoString)

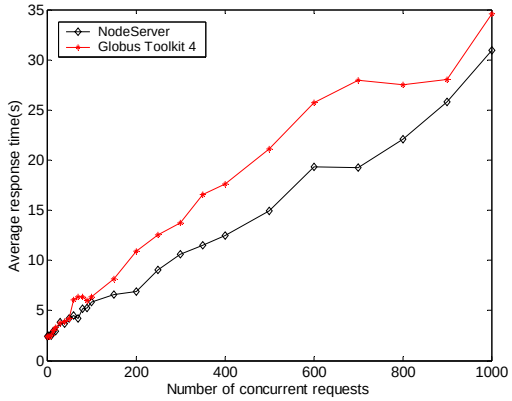

Figure 7. Average response time vs. Num of concurrent requests (echoString)

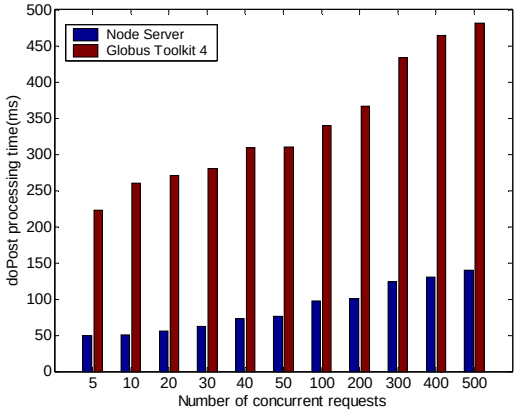

Figure 8. Comparison of doPost function processing time (echoString)

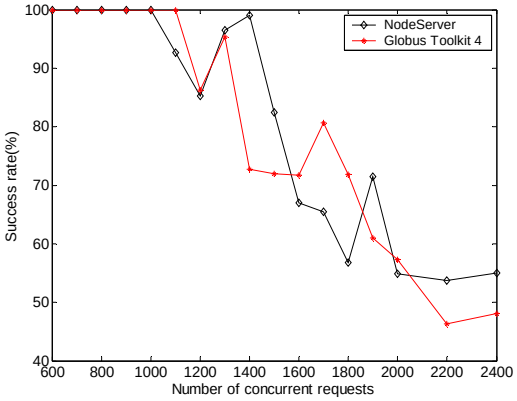

Figure 9. Success rate vs. Num. concurrent requests (echoString)

The results of the first experiment are presented in Figure 6~9. We vary the number of concurrent requests from 1 to 1000. Figure 6 plots the throughput of Node Server and Globus Toolkit 4, while Figure 7 shows the average response time for processing a request. We can see that Node Server outperforms Globus Toolkit 4 almost in all cases, and the performance gap gets bigger along with the increase of concurrent requests. We figure out there can be two reasons to explain the results. First, we redesign the class loader mechanism in Node Server to implement hot service deployment. Each service maintains its own class loader, while GT4 uses one class loader for all

4.2. Evaluation Results

In our first experiment, we hope to evaluate the throughput, average response time and success rate of Node Server and GT4 under different numbers of concurrent user requests. A simple stateless service, *echoString*, is developed as a benchmark service in this experiment. The *echoString* service receives a string from clients, and responds with the same string.

all services. As a result, it will take GT4 more time to load Java classes than Node Server when processing user requests. Second, GT 4 adds several handlers for security processing in both request and response processing chains. Even in non-security mode, the request and response messages will also flow through these handlers, while Node Server removes these handlers in non-security mode.

In the implementation of both GT 4 and Node Server, a request is eventually dealt with by a doPost function that invokes Axis engine to fulfill the processing function. Therefore, on server side, we measure the processing time of doPost function, and

the result is shown in Figure 8. The difference of processing time further verifies that Node Server performs better than GT 4.

Figure 9 plots the success rate of Node Server and GT4. We notice that the results have some fluctuations, but the success rates of both implementations begin to decrease at around 1100 concurrent requests.

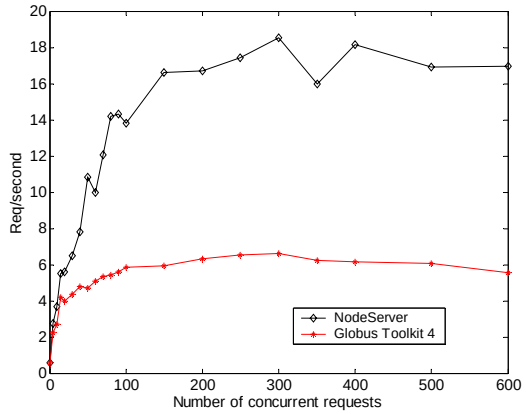

Figure 10. Throughput vs. Num of concurrent requests (counter)

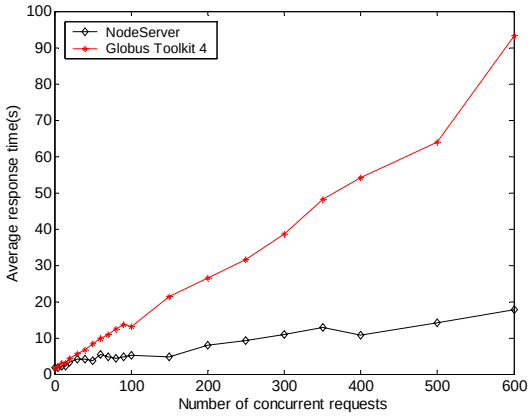

Figure 11. Average response time vs. Num of concurrent requests (counter)

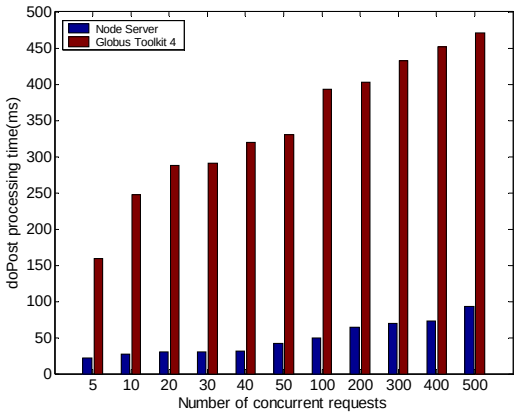

Figure 12. Comparison of doPost function processing time (counter)

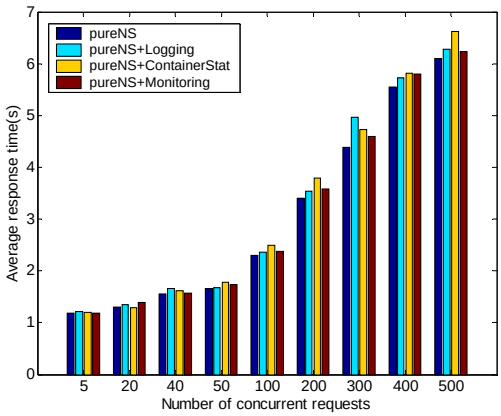

Figure 13. Performance of three base services (echoString)

Figure 10~12 show the results of the second experiment. We can see that similar conclusions to that of the first experiment can be drawn except that processing one *counter* request incurs bigger overhead than processing one *echoString* request. This is because that one counter request involves four interactions with server side: creating a *counter* resource, increasing *counter* value, getting *counter* value and destroying *counter* resource. However, when we try to measure the success rate of the two implementations by varying the number of concurrent requests, the Globus Toolkit turns dead with 500 concurrent requests while works normally with 450 concurrent requests. By now, we do not know the concrete reason behind this.

As it is an important feature to support stateful resources in WSRF implementation, in our second experiment, we measure the throughput, average response time and success rate using a stateful WSRF service, *counter*, which is provided with GT 4 release.

In the third experiment, we evaluate how our base services influence the performance of Node Server through invoking the *echoString* service concurrently. The base services considered here includes Resource Monitoring service, Logging service and ContainerStat service. All three services mentioned above involves a daemon thread that starts to run when Node Server starts, while the deployment service will not incur overhead without deployment requests. The performance of deployment service is evaluated in [11], thus we do not consider it in this experiment. We

evaluate the performance of Node Server in the following scenarios: (1) pureNS; (2) pureNS + Logging;(3) pureNS + Monitoring; (4) pureNS + ContainerStat, where pureNS means pure Node Server environment without installing base services. The results in Figure 13 show that the three base services do not incur too much overhead to Node Server itself.

5. Conclusion

In this paper, we present the design and implementation experience of an enhanced grid service container, CROWN Node Server. Our work leverages existing GT4 Java WSRF core, while we do several important extensions in terms of service deployment, resource status monitoring, logging, container statistics and management. These extensions are included in CROWN 2.0 release that have been tested in CROWN testbed and widely used in real applications as well. We perform extensive experiments to evaluate the performance of our implementation and Globus Toolkit. The results show that Node Server performs better that Globus Toolkit under various numbers of concurrent requests.

We will perform more tests on all components of CROWN Node Server to make it stable to use. Also we hope to further evaluate the performance of the most updated version of Globus Toolkit. In addition, we will perform further study on performance issues.

Acknowledgement

We thank our colleagues, Zhong Zheng, Wei Zhang and Liang Zhong, for their help with client-side programming work and setting up the experimental environment.

This work was supported in part by the NSF of China under Grant 90412011 and by China National Natural Science Funds for Distinguished Young Scholar under Grant 60525209, partly by National Basic Research Program of China 973 under grant no. 2005CB321803, and partly by Development Program of China 863 under grant no. 2005AA119010

References

[1] Globus Toolkit: http://www.globus.org/toolkit/.
[2] Apache Axis: http://ws.apache.org.
[3] OMII: http://www.omii.ac.uk.
[4] CROWN project: http://www.crown.org.cn/en
[5] GLORIAD:http://www.gloriad.org/gloriad/
[6] PRAGMA: http://www.pragma-grid.net/index.htm
[7] Apache Log4J: http://logging.apache.org/log4j
[8] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "Grid Services for Distributed System Integration," IEEE Computer, vol. 35, pp. 37-46, 2002
[9] I. Foster and C. Kesselman, The Grid 2: Blueprint for a New Computing Infrastructure. San Francisco: Morgan Kaufmann, 2003
[10] H. Sun, L. Zhong, J. Huai, and Y. Liu, "OpenSPACE: An Open Service Provisioning and Consuming Environment for Grid Computing," e-Science 2005.
[11] H. Sun, Y. Zhu, C. Hu, J. Huai, Y. Liu, and J. Li, "Early Experience of Remote & Hot Service Deployment with Trustworthiness in CROWN Grid," APPT 2005.
[12] W. H. Winsborough, K. E. Seamons, and V. E. Jones, "Automated Trust Negotiation," in Proceedings of DARPA Information Survivability Conference and Exposition, 2000.
[13]M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation and Experience", *Parallel Computing*, Vol. 30, Issue 7, July 2004.
[14] R. Wolski. "Dynamically forecasting network performance using the network weather service". *Cluster Computing*, 1:119-132,1998.
[15] R. Wolski, N. Spring, and J. Hayes. "The network weather service: A distributed resource performance forecasting service for metacomputing". *Future Generation Computer Systems*, 15(5-6):757-768, 1999