Dependency-aware Maintenance for Dynamic Grid Services[*]

Hai Jin, Li Qi, Song Wu, Yaqin Luo, Jie Dai
Cluster and Grid Computing Lab
Service Computing Technology and System
Huazhong University of Science and Technology
hjin@hust.edu.cn

Abstract

Any mistaken maintenance for the complicated and distributed grid can bring unpredictable disaster. Here we focus on the system availability issues caused by service dependencies during the maintenance in grid. A novel mechanism, called Cobweb Guardian, is proposed in this paper. It provides multiple granularities (service-, container-, and node-level) maintenance for service components in grid. By using the Cobweb Guardian, grid administrators can execute the maintaining task safely in runtime with high availability. The evaluation results show that our proposed dependency-aware maintenance can make the grid management more automatic and available.

1. Introduction

Today's web or grid services are usually composed by several standalone services that encapsulate and present useful functionalities. Examples include simple storage services, online digital album, execution management system, and virtual data centers. Within these complex services, the service components are partitioned, replicated, and aggregated to achieve high availability and incremental scalability, especially when the system experiences high growth in service evolution and user demands.

From the experience of *ChinaGrid Support Platform* (CGSP) [11] and VEGA [12], the scale and complexity of *Virtual Organizations* (VO) [5] make them increasingly difficult and expensive to manage and deploy. A system update at even a moderate scale data center in a VO can require changes to more than 1,000 machines, some of which might have interdependencies among their services. That means any maintenance to a service component must be propagated or contained so that the services using that component continue to function correctly. Furthermore, the availability of global management units for service requests during the maintenance should be maximized. Because services have to be paused in the traditional procedure of software maintenance, however some system such as bank services must provide services continuously in 24 hours, any unpredictable pause will make great lost.

For the grid administrators, the maintenance task is running through the whole lifecycle of service components. As shown in Figure 1, each service component in a grid has the lifecycle of: released, deployed, initialized, activated, and destroyed. Responding to these stages, the maintenance tasks include: *publish, deploy, undeploy, redeploy, configure, activate,* and *deactivate*. Especially, these tasks should face the challenges in grid environment.

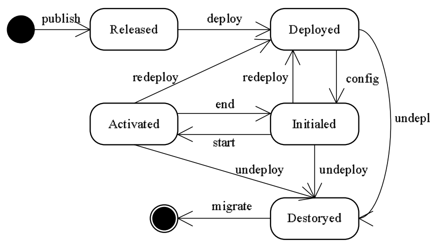

Figure 1. Lifecycle of Service Component

This paper recognizes the importance of distributed service maintenance and its challenges. The main goal of this study is to answer the following question. How to promise the higher availability for a global system when a maintenance operation happens on a service component with the complicated dependencies?

[*] This work is supported by the NSFC under grant 60673174, 90412010 and the National 863 Hi-Tech project under grant No.2006AA01A115

For higher availability during maintenance procedure in dynamic grids, we propose a new mechanism, called *Cobweb Guardian*, which supports multiple granularities (service-, container-, and node-level) maintenance for service components. The *Cobweb Guardian* manages different service dependencies map that is initially provided by the administrator. When a maintenance request arrives, it recognizes the related dependency to generate an optimized maintenance solution and reduce the affects from dependencies. In addition, a *Cobweb Guardian* provides the session management for maintenance to avoid the possible failures by dependency hierarchy.

The rest of this paper is as follows. In section 2 we describe the problem statement and the motivations. In section 3 we present an overview of architecture and some details about design, and implementation. In section 4 we evaluate our approach in different stages. In section 5 we explore the former and related works, and in section 6 we conclude with a brief discussion of future research.

2. Motivations

2.1 Concepts

Before further discussion, we introduce some basic concepts:

- *Service Components* in a grid or web-based system are hosted in some specific *container* such as Globus Toolkit 4 Java WS Core [2] and provide a set of operations in public that can be used to compose some new services. The communications among them are always encapsulated in a *Message* by some protocols such as SOAP. In addition, we define the service that depends on other service components as *Composite Service*.
- The *Dependency* discussed in this paper denotes that the correct execution of a service component is always depending on the hosting environment, the dependent calling services, and the dependent deployment service respectively.
- The *Maintenance* to a grid or distributed web services based system includes the operations (e.g. deploy, undeploy, and so forth) to some particular service components distributed in grid. Normally the maintenance requests are delivered by the administrators.

To state the problem clearly, two metrics are defined for the maintenance.

Maintenance Time. In the distributed environment, the maintenance time is decided by the transferring time of critical packages (e.g. installing packages, patches, or configuration files), deploying time, reloading time (marked as t_r) of essential components or container, and the pending time (marked as t_p) for executing requests or reloading of remote depending components. In particular, the reloading and pending time are issued in runtime. For instance, the maintenance services are definitely unavailable during reloading and pending.

For a specific maintenance, we suppose that the task covers *n* service components distributed in *m* resources. Ideally, the shortest maintenance time is to deploy *n* services to those resources in parallel. The most cost maintenance is to maintain the service components in different resources in serial. The actual maintenance time is as Formula 1.

$$\max_i^n (t^i) < t < m \cdot \sum_i^n t^i \tag{1}$$

t^i means the average maintenance time for the *ith* service in the collection of maintenance task.

Availability. The availability (marked as *A*) that we introduced is *the proportion of time a system is in a functional condition in the watching period.* More specifically, we define the availability (in Formula 2) of the system during the maintenance is the ratio of system's available time to the longest maintenance time (i.e. watching period). In particular, the symbol U means the combination of pending and reloading time instead of the sum of them since the pending and reloading operations can be overlapped during the transferring or deploying of other components.

$$A = 1 - \frac{\bigcup_i (t_p^i + t_r^i)}{\sum_i t^i} \tag{2}$$

It denotes the better solution for maintenance is with less maintenance time and higher availability. To achieve that, some analysis should be done for the different dependencies. We also introduce the *Loss Rate*, the ratio of failure requests to the total requests during the longest maintenance time. Actually, it denotes the inverse proportion to the availability.

2.2 Dependencies in Grid

As mentioned in section 1, service dependency is very complicate in grids. From the viewpoint of development, each service component depends on a bunch of service components in logic. On other side, from the viewpoint of deployment, service components also depend on the target hosting containers. Based on the literature [4][10], and our experience, we conclude the dependencies into three main types.

Invocation Dependency. This kind of dependency represents the trend of SOA that happens among composite services [4][11]. We describe three kinds of invocation dependency as examples. (i) *AND-dependency* describes the aggregating relation of other service components. The typical scenario is the executing system in grid. The execution managers (e.g. *Grid Resource Allocate and Management*, GRAM [2]) are always AND depending on the information systems (e.g. *Monitoring and Discovery System*, MDS [2]) and data file-transfer (e.g. *Reliable File Transferring* service, RFT [2]). (ii) *XOR-dependency* means the replication relations. It can be explained as *switch-case* logic. It is popularly seen in data centers. (iii) *OR-dependency* means that the invocations to related services can be ignored. It can be mapped to the *try…catch…finally* logic. The cache service is its classic use case. Most of web systems provide the caches for front end service. System tries finding cache first and then dispatch to the real database if the former failed.

Deployment Dependency. The deployment of specific services always requires other related services deployed first in target executing container. In addition, the versioning problem during the maintenance also drives us to focus more on this kind of dependency. The typical example is that a deployment of a service component (e.g. wikipedia website) requires a database component in a specific version (e.g. MySQL-server-4.0.20-0) installed in the remote site for initializing purpose. If we consider nothing about this, the maintenance work will fail. The deployment dependency includes the configuring dependency discussed in literature [10].

Environment Dependency. The maintenance for one specific service could jointly affect other services' maintenance in the same environment when some functional services are deployed into the same hosting computational node. For example, if the Data and the Info service have been deployed on a same container, despite they are neither deployment nor invocation depended on each other, when maintaining the Data service, the availability of Info service is also affected because the maintenance requires restarting the node.

2.3 Traditional Maintenance Solution

In this section, the affects from dependencies during maintenance for traditional approaches will be demonstrated. As mentioned in Section 2.1, administrators always maintain the distributed systems in serial (e.g. using the Bash shell scripts in Linux) or in parallel (e.g. using Java or C's threads).

Language-based Maintenance in Parallel. In this solution, if there are no considerations about the dependencies, the maintenance can be finished in one step. However, if there is a deployment dependency, the availability will drop to zero and is unable to recover even after the maintenance. The reason is that the failure of deployment dependency crashes the maintenance to the depending service component while the depended services are maintained as normal. The result is unique that the system is unavailable forever. Meanwhile, this approach is not suitable for invocation dependencies (e.g. OR- and XOR-dependency) either. Because whatever the composite service or its service components finish maintenance firstly, the availability of the system is sacrificed. Actually, the availability could be higher if we just maintain the service components one by one.

Script-based Maintenance in Serial. As the other frequently used approach for the daily maintenance, it finishes the maintenance one by one in sequence. Although the serial approach can resolve the deployment dependency problem, it is not acceptable for most maintaining cases for its poor availability and efficiency.

2.4 Objectives

The shortages of the traditional maintenance approaches bring the inconvenience for administrating the large scale service grid when introduce the service dependencies. The objectives of our design include: (i) improve the global availability during the maintenance, (ii) reduce the possible failures of maintenance, and (iii) improve the efficiency of maintenance.

3. Design of Cobweb Guardian

3.1 Maintenance Granularity

Maintaining the complicated dependencies among the services and hosting environment in a grid is a huge engineering work. The efficient reduction of maintenance granularity can help improving the efficiency of maintenance. Three-layer architecture is proposed in Cobweb Guardian to reduce the affects from dependency hierarchy.

1. Service Level. It exists as a manager in the target hosting environment. The service-level maintenance means that all the maintenance tasks are for service component. The reloading or pending operations are also executed as the unit of services.

2. Container Level. It also exists in the hosting environment. Unlike service-level maintenance, the reloading or pending operations are for the whole container. Namely, if a service in a container needs the maintenance, the other services are also put into the maintaining states.

3. Node Level. The maintenance in node level is from the global view. The reloading or pending operations in this level will issue a bunch of computing nodes. It balances the maintaining policies among the different containers. By communicating with the service- and container-level maintenance manager separately, it can promise that the optimized and safe maintenance solutions are adopted. For instance, we can avoid the affects from environment and deployment dependencies.

3.2 Architecture

As shown in Figure 2, the Cobweb Guardian is a two-tier model, the *Cobweb Guardian (CG)* and *Atomic Guardian (AG)*. A CG communicates with multiple AGs to execute the node-level maintenance for the grid. It is composed of four main functional modules: Session Control, Dependency Optimizer, Authorization center, and Policy controller.

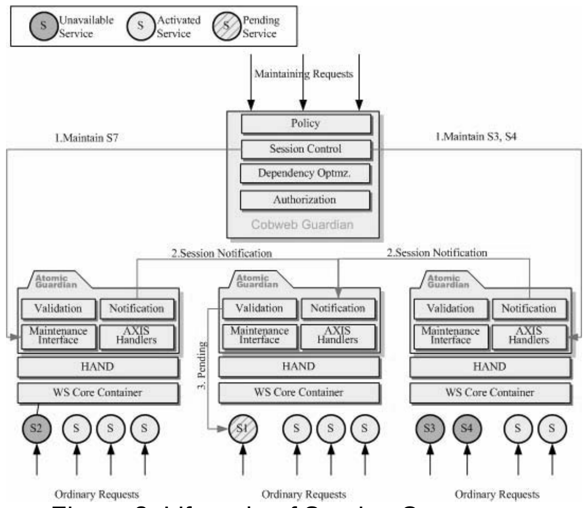

Figure 2. Lifecycle of Service Component

The Policy module is designed for administrator to execute on demand maintenance. The Session Control module is mainly in charge of the progress of the maintenance. In addition to that, it also propagates the maintenance tasks to the target replica Atomic Guardian. Dependency Optimizer is the kernel of CG. It parses administrator's input and matches the requirements to existing dependency maps. Meanwhile, the Authorization module is designed for checking all the maintenance requests to avoid unpredictable dangers (e.g. the requests to deploy Trojan viruses).

The Atomic Guardian, the actual maintainer, is implemented based on our former works [8]. By invoking AG, the Guardian system can support container- and service-level maintenance. AG also consists of four parts.

The Notification module reports the maintenance states to depending services and the Cobweb Guardian. With this module, administrators can grasp the progress of whole maintenance and detect the failures at the first time. The notifications would only be sent to the peer which depending on the service under maintenance. The Validation module is the sink of peer container's Notification module. AG will execute pending or policy defined actions to issue the corresponded notifications. The Maintenance Interface accepts the requests from Session Control. It actually executes the deploying, upgrading, or activating works by talking with the management module of hosting container, e.g. HAND [8]. The Axis handler is designed to record the different invocation dependencies from the input and output message flow. Any recorded peers would be notified by the Notification module during the maintenance. It can efficiently help CG reducing the overhead.

3.3 Propagating Maintenances with Session Control

To avoid the affects from the deployment dependency, the CG will check it before the maintenance and generate the critical maintenance path to propagate the operations. Namely, CG provides the session mechanism to promise the procedure of maintenance. The idea is from the parallel dynamical maintenance for *k*-connectivity graph [6]. CG propagates the maintenance operations of depended service first to promise the maintenance be executed correctly. If the propagations are finished in *k* steps, this solution will cost time of:

$$t = \sum_{i}^{k} \max_{j}^{s(i)} (t^j) \tag{3}$$

s(i) in formula 3 means the parallel maintenance for step *i*. The availability is:

$$A = 1 - \left(\sum_{i=0}^{k} (t_r^i + t_p^i)\right) \Big/ \sum_{i=0}^{n} t_i) \tag{4}$$

It denotes that the maintenance time is longer than simple parallel solution mentioned in section 2.3. It promises the correctness of maintenance.

3.4 Grouping Maintenance for Invocation Dependency

To achieve higher availability during the maintenance, the different semantic of invocations should be differentiated. (i) For AND-dependency, the maintenance for any service in this dependency will reduce the availability since the unavailability of any component can be transmitted in chain to the whole system. The best solution is to propagate all of maintenances to the target containers at the same time. (ii) For XOR- and OR-dependency, because the ordinary requests are always dispatched to the composite services in certain semantic (e.g. round robin, random, or load balanced policy), if the maintenance for composite services can be executed in groups, the availability can be improved to some extend. Formula 5 describes the improved availability by using this approach. p_i ($0<p_i<1$) means the possibility of dispatching requests to the maintenance containers. In addition, t^0 means the maintenance to the front end service component.

$$A = 1 - (t_r^0 + t_p^0 + \sum_{i=1}^{k} p_i \cdot (t_r^i + t_p^i)) \Big/ \sum_{i=0}^{n} t_i) \quad (5)$$

3.5 Grouping Maintenance with Feedback

Although the grouping solution improves the availability by sacrificing the maintenance time, it also brings the unpredictable factors to the system. For instance, some critical invocations will be rejected randomly. CG also provides the feedback notifications for the grid applications. By checking the status of remote service component, the grid applications can bypass the requests to the maintenance service component. In this solution, the availability can be improved much. Actually, the cost is mainly from the maintenance of composite service component.

$$A = 1 - ((t_r^0 + t_p^0) \Big/ \sum_{i=0}^{n} t_i) \quad (6)$$

4. Evaluations

We experiments with our implementation on ChinaGrid test bed. Services in our experiments are evaluated on *ChinaGrid Support Platform* v2.0.1 [11].

Our evaluation has the following objectives: (i) demonstrate the improved availability by comparing the traditional maintenance approach with Cobweb Guardian; (ii) compare the availability and throughput of different maintenance solutions of Cobweb Guardian when the maintenance happens in different dependencies; (iii) demonstrate the effectiveness of dependency feedback and different granularity in improving service availability and throughput.

4.1 Test Environment

Unless stated otherwise, experiments in this paper are conducted on two rack-mounted Linux clusters: one is with 16 1GHz Pentium III nodes (each with 512MB memory). Each node runs Redhat Linux with kernel version 2.4.20-8. The Java runtime version is J2SDK 1.5.0_06-b05 implemented by SUN. The other one employs 20 dual 1.3GHz Itanium2 servers. The nodes inside the cluster are connected by a 100Mbps Ethernet switch. Each node runs Redhat Linux with kernel version 2.4.0-2. The Java runtime version is J2SDK 1.5.0_03-b07 implemented by BEA. The two clusters are also connected with bandwidth of 100Mbps.

Abbreviations. Here is a list of the abbreviations that we will use in the rest of this section: (1) NonD denotes the maintenance solution without dependency consideration; (2) SRL maintains the services by calling shell scripts in serial; (3) CG-0 is the simple propagating solution without optimization for invocation dependency; (4) CG-1 means the grouping maintenance for different invocation dependencies; (5) CG-2 is the grouping maintenance with feed back; (6) REQ in the diagram denotes the fixed requesting rates for particular service components.

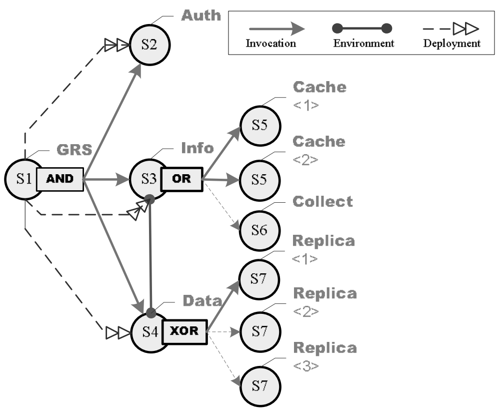

Figure 3. Service Dependencies in CGSP's Execution System

Application for benchmark. Our experiments are based on the typical executing system, called *General Running Service* (GRS), from CGSP. As shown in Figure 3, GRS includes six service components: authentication service (*Auth*), information service (*Info*), data management service (*Data*), cache service (*Cache*), collecting service (*Collect*), and replication service (*Replica*). When a request arrives, GRS parses it and then contacts with the *Auth* component to check the validity, the *Info* component to get job's executing information, and the *Data* component to fetch the staging data, respectively. More particularly, the *Info* component will fetch the job information from the *Cache* firstly, if failed then invoke the *Collect* to select in backend database. On the other hand, the *Data* adopts a replication to store and load staging data. There can be multiple partitions for *Info* and GRS. The detailed semantic about CGSP's execution system can be investigated in literature [11]. By injecting the maintenance to these services, we can inspect the capability of Cobweb Guardian.

4.2 Propagation and Deployment Dependency

To investigate the efficiency of the propagation feature and the affects from deployment dependency, In this experiment, we drive the GRS service at 6 requests per second. We run the experiment for 140 seconds. At second 15, we inject the maintenance to the GRS (including upgrades of *Auth*, *Info*, *Data* service components, and itself) which then becomes unresponsive for a while. We try three solutions: NonD, SRL, and CG-0 in this experiment.

Figure 4 shows the throughput of three solutions during the 30-second watching period. When there is no maintenance, all systems work well. When maintenances start at second 15, the throughputs of the system with different approach are all falling down to zero. Although the NonD solution finishes the maintenance earliest, it can not work correctly any more. Because it does not consider the deployment dependencies among the service components, the maintenance for GRS component fails. The SRL solution costs longest time to finish the maintenance. CG-0 provides the highest efficiency. It loses 49.1% requests because the AND-depended service components are unavailable when any related service (i.e. *Auth*, *Info*, and *Data*) is under the maintenance.

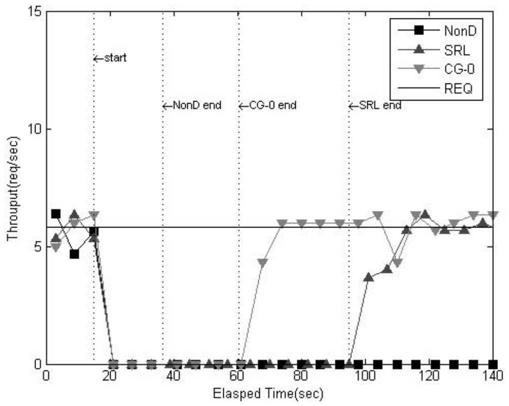

Figure 4. Throughputs of Execution Service during Maintenance for Different Approaches

4.3 Optimization for Invocation Dependency

In the next two experiments, we use the Info service to evaluate the effectiveness of Cobweb Guardian for improving service availability upon different invocation-dependencies. We drive the system at 10 requests per second to Info service.

OR-Dependency. As shown in Figure 5, at second 15, we start maintenance solutions CG-0 and CG-1, respectively. CG-0 propagates the maintenance tasks to the target containers that deployed Cache and Collect service components and then to the one with Info service. The whole procedure costs 62.97 seconds. However, from Table 1, we can find that the throughput during executing CG-0 is falling down near to zero. In addition, there are 57.9% requests are lost in the watching period (defined in section 2).

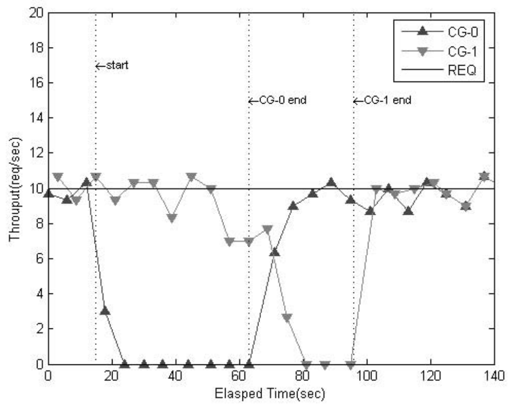

Figure 5. Maintenance for the Information Service

TABLE 1 COMPARISON OF EXPERIMENTAL RESULTS

Appr	Maintain	RespTime (*ms*)	Throughput (*req/s*)	LossRatio (*%*)
CG-0	Before	443.9	9.51	0
	During	N/A	0.86	57.9
CG-1	Before	463.8	9.92	0
	During	929.8	7.08	25.6

Compared with CG-0, CG-1 optimizes the solution for OR-dependency. The Cobweb Guardian maintains the target containers one by one in groups with different priorities. The improvement is obvious: first, the throughput during the maintenance is improved to 7.08 requests per second. Although the average response time is 929.8 ms which is about double of that before maintenance, the loss rate is reduced from 57.9% to 25.6%. Meanwhile, the 25.6% is the minimum cost for all solutions because the maintenance for Info service is the key maintenance and is hard to avoid. From the figure, we can identify that the throughput of CG-1 is falling down to about 7.3 from second 57. The reason is that the requests to Cache services are failed when Cobweb Guardian are maintaining them. The requests are sent to the Collect service forcedly.

XOR-Dependency. We repeat the experiment for Data service component (XOR-dependency). This time we add solution CG-2. The maintenance solution will feed back the maintenance status to the user-level. Similarly, we start the three maintenances at second 15.

Figure 6 describes the whole procedure: The CG-0 solution acts similarly like before. It blocks all the requests to the Data service during the maintenance. Hence the throughput for CG-0 is lowest. However the CG-1 solution acts also not well. Although it improves the throughput to some extend (from 0 to 4.71), the loss rate is about 52.1% which is a minor improvement to CG-0's 53.9%. The main reason is that CG-1 costs more time to finish the maintenance than CG-0 but the Data service is always trying to deliver the requests to the Replicas during maintenance. These requests are failed definitely. Unlike CG-0 and CG-1, CG-2 works far better. It costs same time to finish maintenance like CG-0. However it gives better throughput (6.56), lower loss rate (25.9%), and better response time for normal requests (478ms).

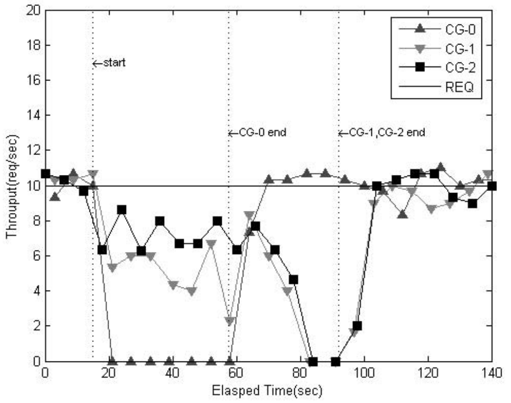

Figure 6. Throughputs of the Virutal Data Center Service during Maintenance

Table 2 lists the response time, throughput, and loss rate respectively for the three solutions in different stages (before and during) of maintenance. It reflects the variations discussed above. The results prove that Cobweb Guardian can find the best solution for different dependencies.

TABLE 2 COMPARISON OF EXPERIMENTAL RESULTS

Appr	**Maintain**	**RespTime (*ms*)**	**Throughput (*req/s*)**	**LossRatio (*%*)**
CG-0	Before	552.5	9.21	0
	During	N/A	0.01	53.9
CG-1	Before	405.6	9.43	0
	During	933.2	4.71	52.1
CG-2	Before	405.1	10.07	0
	During	478.5	6.56	25.9

4.4 Optimization for Environment Dependency

To demonstrate the enhancement on environment dependency, we execute the upgrade for Info service component deployed with Data service component in a same container. Figure 7 describes the results. *A* denotes container-level maintenance and *B* is service-level. In the container-level maintenance, the Data service is also unavailable when Info service is under upgrading. However, the maintenance in service level does not affect the accessing to Data service. In addition, the maintenance time in service-level (14.6 sec) is also shorter than that in container-level (23.6 sec). This result proves that the availabilities for Data and Info service components are enhanced.

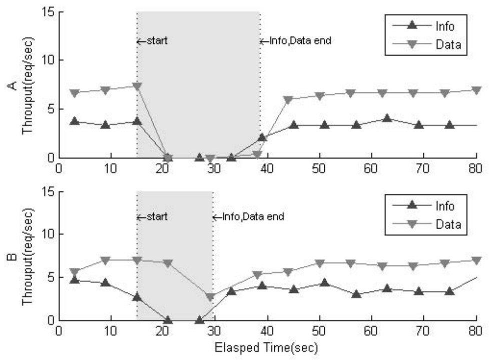

Figure 7. Maintenance in Different Gruanularities for Environment Dependency

5. Related Works

Neeraj et al [9] invented a dependency structure matrix to describe the dependencies among legacy software. It is efficient for exploring the software architecture from the viewpoint of software engineering. But they did not discuss the dynamic dependencies which are common in distributed and dynamical grids. The dependency structure matrix is not convenient for dynamic deployment in grid.

Service capsule proposed by Lingkun et al. [4] is a new mechanism to support automatic recognition of dependency states and per-dependency management for thread-based services. Nevertheless capsule focus more on fault tolerant instead of maintenance. In addition, it works for the multi-threading cluster servers and can not process complicated dependencies.

System availability is an important issue for distributed systems, which has been addressed extensively in the literatures [4][8][10]. Typical metrics for measuring the overall system reliability are MTBF (*mean time between failures*) and MTTR (*mean time to recovery*). It often takes a long period of time to measure these metrics. Recently, the fault injection has been proposed as an effective but less time-consuming means to assess the system availability [7].

In our former work [8], the two approaches (named service- and container-level) of dynamic deployment were proposed to enhance the availability of service infrastructure. The result proved that choosing the smaller granularity deployment (in service-level) can improve the system availability to some extend. However, whatever service- or container-level, the improvement is limited in infrastructure layer. It can not promise the global availability with invocation dependencies among the services.

The *Configuration, Description, Deployment and Lifecycle Management* (CDDLM) specification [1] proposed by GGF and the *Installable Unit Deployment Descriptor* (IUDD) [3] proposed by W3C are all the specifications to standardize the maintenance works for distributed software or services. But the two specifications do not promise the quality of maintenance operations and the runtime availability during the maintenance.

Vanish et al [10] compared manual, script-, language-, and model-based deployment solutions in terms of scale, complexity, expressiveness, and barriers for distributed services. Despite the dependency problem discussed, the affects to the global availability during deployment was not discussed.

6. Conclusion and Future Works

In this paper, we propose the Cobweb Guardian which is dependency-aware maintenance architecture for grids. By investigating affects from the different dependencies in the runtime of grid (including invocation-, deployment-, and environment-dependency), the Cobweb Guardian can automatically generate the optimized solutions for the maintenance in distributed grids. The evaluations demonstrate the effectiveness of the Cobweb Guardian to improve the availability and throughputs during maintenance.

The further works include the investigation on the fault tolerance for distributed maintenance because the cost from failed maintenance affects the availability of system. In addition, the QoS challenge in runtime is another important research issue.

Reference

[1] Configuration, Deployment Description Language and Management, GGF, http://www.gridforum.org/documents/GFD.50.pdf

[2] Globus Toolkit Project, Globus Alliance, http://www.globus.org

[3] Installable Unit Deployment Descriptor Specification, http://www.w3.org/Submission/InstallableUnit-DD/

[4] L. Chu, K. Shen, H. Tang, T. Yang, and J. Zhou, "Dependency isolation for thread-based multi-tier Internet services", *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, March 2005, Miami, FL, USA, pp.796-806.

[5] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International J. Supercomputer Applications*, Vol.15, No.3, 2001.

[6] W. Liang, R. Brend, and H. Shen, "Fully dynamic maintenance of *k*-connectivity in parallel", *IEEE Trans. on Parallel and Distributed Systems,* Vol.12, No.8, Aug 2001, pp.846-864

[7] K. Nagaraja, X. Li, B. Zhang, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services", *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, Mar. 2003.

[8] L. Qi, H. Jin, I. Foster, and J. Gawor, "HAND: Highly Available Dynamic Deployment Infrastructure for Globus Toolkit 4", *Proceedings of the 15th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Naples, Italy, Feb. 2007.

[9] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using Dependency Models to Manage Complex Software Architecture", *Proceedings of OOPSLA'05,* 2005, pp.167-176.

[10] V. Talwar, D. Milojicic, and Q. Wu, C. Pu, W. Yan, and G. Jung, "Approaches for Service Deployment", *IEEE Internet Computing*, Vol.9, No.2, Mar 2005, pp.70-80.

[11] Y. Wu, S. Wu, H. Yu, and C. Hu, "CGSP: An Extensible and Reconfigurable Grid Framework", *Proceedings of the 6th International Workshop on Advanced Parallel Processing Technologies*, Hong Kong, China, October 27-28, 2005, pp.292-300.

[12] Z. Xu, W. Li, L. Zha, H. Yu, and D. Liu, "VEGA: A Computer Systems Approach to Grid Computing", *Journal of Grid Computing*, Vol.2, No.2, 2004, pp.109-120.