The Journal of Systems and Software 83 (2010) 1983–1994

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Dependency-aware maintenance for highly available service-oriented grid

Hai Jin[a,*], Yaqin Luo[a], Li Qi[a,b], Jie Dai[a], Song Wu[a]

[a] Huazhong University of Science and Technology, Wuhan, 430074, China
[b] China Development Bank, Beijing, 100037, China

ARTICLE INFO

Article history:
Received 12 November 2007
Received in revised form 26 May 2010
Accepted 26 May 2010
Available online 16 June 2010

Keywords:
Dynamic maintenance
Grid services
Service dependency
Availability

ABSTRACT

When the scale of computational system grow from a single machine to a Grid with potentially thousands of heterogeneous nodes, the interdependencies among the resources and software components make management and maintenance activities much more complicated. One of the most important challenges to overcome is how to balance maintenance of the system and the global system availability. In this paper, a novel mechanism is proposed, the Cobweb Guardian, which provides solutions not only to reduce the effects of maintenance but to remove the effects of dependencies on system availability due to deployment dependencies, invocation dependencies, and environment dependencies. By using the Cobweb Guardian, Grid administrators can execute the maintenance tasks safely at runtime whilst ensuring high system availability. The results of our evaluations show that our proposed dependency-aware maintenance mechanism can significantly increase the throughput and the availability of the whole system at runtime.

1. Introduction

Today's web or Grid services are usually composed by several standalone services that present useful functionalities. Examples include (i) online digital album that is composed by simple storage services and metadata management services; (ii) execution management system that consists of information services, authentication and authorization services, data transferring services, and so forth. Within these composite services, the service components would be partitioned, replicated, and aggregated to achieve high availability and scalability, especially when the system experiences high growth in service evolution and users' demands. However from the experience of *Chinagrid Support Platform* (CGSP) (Wu et al., 2005) and VEGA (Xu et al., 2004), the incremental scale and complexity of *Virtual Organizations* (VO) (Foster et al., 2001) make the management and deployment increasingly complicated and much more difficult to execute. Even at a moderate scale data center, a system update might affect more than 1000 machines, since most of them are interdependent on each other (Talwar et al., 2005). It means that the maintenance to a service component should be carefully contained to ensure the continuously correct execution of affected services. Further more, the availability of management components during the maintenance needs to be kept in a comparatively high level in order to continually react to the service requests. In traditional distributed systems, the related services might have to be shut down temporally during the maintenance. However some critical commonly used business services such as Bank services must provide services continuously in 24 hours per day since any unpredictable pause from the maintenance will lead to a big lost.

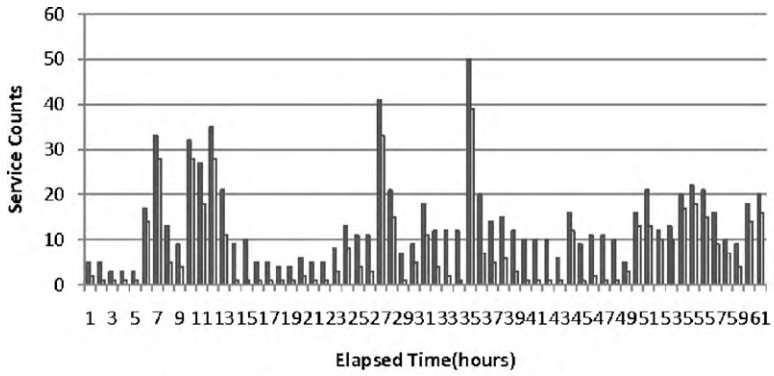
Specifically, for Grid administrators, the maintenance for services is running through the whole lifecycle of service components. Each service component in Grid has five status: *released*, *d* eployed, *initiated*, *activated*, and *destroyed* (Hai Jin et al., 2007). According to these status, the maintenance tasks include *publish*, *deploy*, *undeploy*, *upgrade*, *configure*, *activate*, and *deactivate*. How to coordinate these maintenances without sacrificing the utilization of target resources and the availability of the whole system is indeed a great challenge.

As shown in Fig. 1, we take the realistic logs of CGCL (a domain of ChinaGrid) as an example. During the observed 60 h (from 00:00:00 GMT+8 Jul 18, 2008 to 12:00:00 GMT+8 Jul 20, 2008), the Grid infrastructure with 41 HPC nodes[1] was continuously serving the requests from ten users(Fig. 1(c)). Obviously, it is impossible to complete a upgrade request of infrastructure services (e.g. transferring services, authentication services, and so forth) since there is

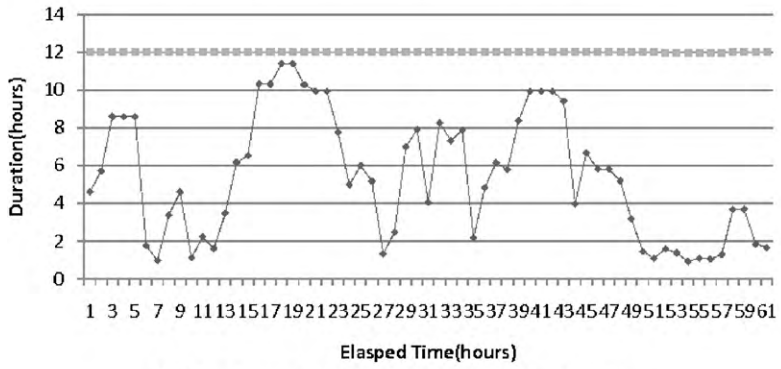
* Corresponding author.
E-mail address: hjin@hust.edu.cn (H. Jin).

[1] The detailed configuration can be found at http://grid.hust.edu.cn/hpcc/index.php.

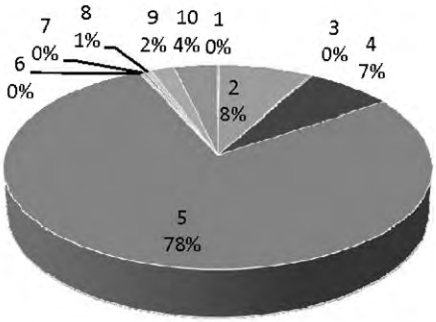
0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.05.068



(a) Services concurrently running in the domain



(b) Services executing duration in the domain



id	user	total(sec.)
1	bhhuang	1402
2	dhpet	112212
3	huangjiao	2346
4	hzhang	103245
5	klyao	1084012
6	kxiao	3427
7	lb	4282
8	test	7210
9	vast	28468
10	zhshao	52195

(c) top 10 users

Fig. 1. Service running statistics in a domain of ChinaGrid.

no time slot for this request. In addition, all user applications were running upon those critical infrastructure services. There would be 33 running services and 28 new requests failed (as shown in Fig. 1(a)) if we run the upgrade at time slot 7 (as shown in Fig. 1(b)) during which there were fewest running requests when we take sub-optimal solution. Therefore, the traditional maintenance can significantly affect system availability. Further more, more requests will be rejected due to the maintenance if the upgrade cannot be finished in an hour.

From this scenario, a sophisticated maintenance mechanism is needed to facilitate administrator in many ways. First, administrator needs not to issue the maintenance tasks for each service component in serial each time. Next, the maintenance mechanism should take the invocation dependencies during runtime into account to choose solutions that can provide higher availability. More over, multiple maintenance granularity should be supported in maintenance mechanism to reduce the effects from environment. Finally, the deployment (or versioning) dependencies between services must be detected in advance before the maintenance, so to reduce the need of making changes and the possibility of making errors to specific maintenance.

The dependencies technologies have been investigated since desktop computing (Yau and Tsai, 1987; Sangal et al., 2005). They are widely used for optimizing in software engineering area. However, the static and simple approaches cannot guarantee system efficiency and availability at runtime as services components distributed widely in Grids. This paper recognizes the importance of distributed service maintenance and its challenges. The main goal of our study is to answer the following questions. How to guarantee high availability of the whole system when a maintenance applied to some service components with complicated dependencies, especially for the essential services?

In this paper, we propose a new maintenance mechanism, named *C* obweb Guardian, which supports multiple granularities (service-, container-, and node-level) maintenance to reduce the effects from environment dependencies. In addition, the *Cobweb Guardian* manages a dependency map that is initially provided by automatic detector or Grid administrators. When a maintenance request arrives, the Cobweb Guardian recognizes the related dependencies using dependency map to generate an optimized maintenance solution which can reduce the dependency effects. Further more, *Cobweb Guardian* provides the session management for maintenance to avoid the possible failures brought by dependency hierarchal.

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 details about design, and implementation. In Section 4 we evaluate our approach by different test cases. In Section 5 we explore the related works including our former efforts, and in Section 6 we conclude this paper with a brief discussion of future research.

2. Motivations

2.1. Concepts

Before further discussions some basic concepts are introduced:

- *Service Components* in a Grid or web-based system are hosted in some specific *container* such as Globus Toolkit 4 Java WS Core (Foster, 2006) and provide a set of operations to public

that can be used to compose new services. The communications between them are usually encapsulated in a *Message* by some protocol such as SOAP. In addition, we define the service that is composed by some other service components as *Composite Service*.

- *Dependency* in this paper denotes a kind of relationship between the service components and hosting environment. The correct execution of a service component is always depending on the hosting environment, the dependent calling services, and the dependent deployment service.
- *Maintenance* to a Grid or a distributed web services based system denotes a set of tasks (e.g. deploy, undeploy, and so on) to some particular service components distributed in Internet-connected computing resources. Normally these requests are delivered by the administrators. However by invoking the maintenances automatically, the Grid system is capable of self-healing, self-growing, and dynamical provisioning.

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

2.1.1. Maintenance time

In the distributed environment, the maintenance time (marked as t) for specific service component (r) is decided by the time of transferring maintenance related packages (t_t, e.g. installing packages, patches, or configuration files), the time of deploying these packages (t_d), the time of pending invocation requests (t_p) due to dependencies, and the time of reloading container or essential components (t_r). Among them, the pending and reloading periods are rather dynamical during the maintenance. Meanwhile most components would be unavailable during these two periods. Formula (1) denotes this equation.

$$t(r) = t_t + t_d + \sum_{s}^{S_r}[t_p(s) + t_r(s)] + t_r(r) \tag{1}$$

In Formula (1), the Sr is the set of service components on which the maintained service component r is depending. The $\sum_{s}^{S_r}[t_p(s) + t_r(s)]$ denotes that the pending time and reloading time of r is decided by the related components recursively.

We suppose that the task covers n service components that distributed in m resources ($n \geq m$). In ideal, the shortest maintenance time (lower bound) obtained is to deploy these service components to the resources in parallel. And the longest maintenance time (upper bound) obtained is to maintain these service components in different resources in serial and with the possible penalty. The actual maintenance time is exposed in Formula (2).

$$\max_i^n(t^i) \leq t \leq m \cdot \left(\sum_i^n t^i + p \cdot t_{pen}\right) \tag{2}$$

The t^i means the average maintenance time for the ith service in the target collection of maintenance task. And the p is the failure possibility of maintenance while t_{pen} is the penalty caused by the maintenance failure.

2.1.2. Availability

The availability (marked as A) that we defined in this paper is *the proportion of available time at which the system is and be capable of executing correctly*. More specifically, we define the availability (in Formula (3)) of the system during the maintenance is the ratio of system's available time to the longest maintenance time (i.e. watching period). The symbol $\bigcup$ in Formula (3) means the combination of pending and reloading time instead of the sum of them since the pending and reloading period might be overlapped during the transfer or deployment of these service components.

$$A = 1 - \frac{\bigcup_i (t_p^i + t_r^i)}{\sum_i t^i} \tag{3}$$

From the two metrics above, we can find that a good maintenance solution is usually coming with less maintenance time and higher availability. To achieve that, we will do the analysis for different dependencies in the next sections. For evaluation convenience, we also introduce the *Loss Rate*, the ratio of failure requests to the total requests during the watching period. Actually, it appears as the inverse proportion to the availability.

2.2. Dependencies in Grid

As mentioned in Section 1, service dependency is common and complicated in service-oriented Grids. From the view of Grid developers, the composite service always depends on a bunch of other service components. On the other side, from the view of Grid administrators, the correct deployment or maintenance of service components also depends on the target hosting containers. Based on the literature (Chu et al., 2005; Talwar et al., 2005) and our experience (Qi et al., 2007), we classify the dependencies into three main types: deployment dependency, invocation dependency and environment dependency.

Deployment dependency: The successful pre-deployment of related services is always necessary basis of a service deployment. In addition to that, the version problem during the maintenance also drives us to focus more on the dependencies between the pre-deploying components and deployed service components. In desktop systems, the Berkeley DB3 is used to record the deployment dependencies for Linux's RPMs (Mugler et al., 2005). In distributed computing environment, the deployment of a service component (e.g. Wikipedia website) requires a database component of a specific version (e.g. MySQL-server-4.0.20-0) must be pre-installed in the remote site for initialization. If Grid administrator ignores this dependency, the maintenance will fail. The configuring dependency discussed in the literature (Talwar et al., 2005) can be included in deployment dependency.

Invocation dependency: It represents key feature of SOA that usually exists between composite services (Chu et al., 2005; Wu et al., 2005). We describe three kinds of invocation dependency as examples:

- *AND-dependency* describes the aggregating relation of service components. The typical scenario is the executing system in a Grid. The execution management (e.g. *Grid Resource Allocate and Management*, GRAM Foster, 2006) is AND depending on the information systems (e.g. *Monitoring and Discovery System*, MDS Foster, 2006), data file-transfers (e.g. *Reliable File Transferring service*, RFT Foster, 2006), and some other system functional services.
- *XOR-dependency* means the replication relation. It can be explained as *switch-case* logic in Java. The requests to composite service will be delivered randomly to the depending components. This kind of dependency is common in today's Data centers. For instance, the Data metadata manager is usually dispatching the requests to the Replica services.
- *OR-dependency* denotes that the requests to some composite services could be ignored in some specific cases. This kind of dependency can be mapped to the *try... catch... finally* logic in Java. For example, the front end service usually choose cache service for information system first, however it could be trans-

ferred to real data base when reading cache failed. That is, front end service tries to dispatch request to real database querying when reading cache failed. In the above scenario, the front end service OR-depends on cache service and database querying service.

Environment dependency: The maintenance for a service might jointly affect other services' maintenance when some functional services are hosted by same computational node, For example, 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 doing the Data service maintenance, the availability and throughput of Info service is jointly affected due to the restart task of the container which was delivered by the maintenance of Data service.

2.3. Maintenance solution and dependencies

We introduced the metrics and three dependencies in Section 2.1 and 2.2. And in this section, we will use these metrics for evaluating current solutions and approaches for distributed maintenance.

Till today, the most popular approaches for distributed maintenance is still built on shell scripts (e.g. Bash shell or Perl scripts) and executed in serial. Typical system includes OSCAR (Oscar), Beowulf (Beowulf introduction and overview), and so on. With performance demands growing, the maintenance solution is being implemented in advanced parallel program. The SmartFrog (Smart frog project) proposed by HP, ProActive (Proactive project) proposed by INRIA, and MPICH-G2 (Karonis et al., 2003) proposed by Globus Alliance are ever adopted to implement parallel maintenance toolkit. There are two types of solutions which have different features of maintenance time and availability.

2.3.1. Language-based maintenance in parallel

By using this solution, we can get maintenance time easily if we ignore all dependencies:

$$t = \max_{i}^{n}(t^i) \tag{4}$$

In addition, the availability of the system can be calculated as:

$$A = 1 - \frac{\max_{i=1}^{n}(t_r^i + t_p^i)}{\sum_{i=0}^{n} t^i} \tag{5}$$

However, if there is a deployment dependency, the availability will drop to zero and will be unable to recover even after the maintenance completed. The reason is that the ignore of deployment dependency will lead to the failure of depending service component while the depended services are maintained as normal. The whole system becomes unavailable since then.

On the other hand, this approach is unacceptable either if there are invocation dependencies (e.g. OR- and XOR-dependency). Because whatever the composite service or its service components finish maintenance first, the availability of system will decline.

2.3.2. Script-based maintenance in serial

As another frequently used approach for daily maintenance, the script-based maintenance in serial cost much more time to finish the maintenance compared to parallel one (As shown in Formula (6)).

$$t = \sum_{i}^{m} t^i \tag{6}$$

And the availability can be presented as:

$$A = 1 - \frac{\sum_{i=0}^{m}\left(t_r^i + t_r^p\right)}{\sum_{i=0}^{n} t^i} \tag{7}$$

In Formula (7), the m means the maintenance steps to be executed for n services. It depicts that the availability during this solution is lower than the parallel one.

Although the serial approach does not lead to deployment dependency problem, it is still unacceptable in most maintenance cases for its poor availability and efficiency.

2.4. Objectives

Section 2.3 discusses the shortages of traditional maintenance approaches since the service dependencies must be taken into account. The motivations to build a highly available maintenance mechanism include:

2.4.1. Improve the global availability during the maintenance

The executions of upgrade, undeploy, and deactivate tasks are inevitable to make some service components inaccessible. It can definitely bring chained reactions to the components that are invocation-depending on them. And finally lower down the availability of the whole system. Also, the environment dependency is another factor to lower down the availability of the whole system. Grid administrator needs a effective mechanism to balance the global availability and demands of maintenance.

2.4.2. Reduce the possible failures of maintenance

As discussed in Section 2.1, the maintenance will failed if the deployment dependency has been ignored. Especially in the highly distributed Grid, the failure will be jointly amplified to the whole system and make it crashed down eventually. The traditional maintenance approaches (e.g. manual, script-, and language-based in Talwar et al., 2005) cannot resolve this problem well without handling the dependencies. On the other hand, when a service is being deployed or upgraded, the maintenance will fail if the target environment lacks its depending service packages (i.e. environment dependency). To avoid that, the maintenance mechanism should be able to find out these depending components and deploy them first.

2.4.3. Improve the efficiency of maintenance

As mentioned in Section 1, most of maintenance tasks can be propagated to the related resources in parallel to reduce maintenance time. We try to find out an optimized solution to balance the parallelism and correctness when introduce the dependency factors.

3. Design of Cobweb Guardian

In this section, we will demonstrate our design of maintenance mechanism to shield the effects from various dependencies. The mechanism in our design has a two layered architecture which can efficiently record and parse dependencies automatically.

3.1. Architecture

As shown in Fig. 2, the two layers are the *Cobweb Guardian (CG)* and *Atomic Guardian (AG)*. A CG communicates with multiple AGs to execute the maintenance tasks for the Grid. Cobweb Guardian is composed of four function modules:

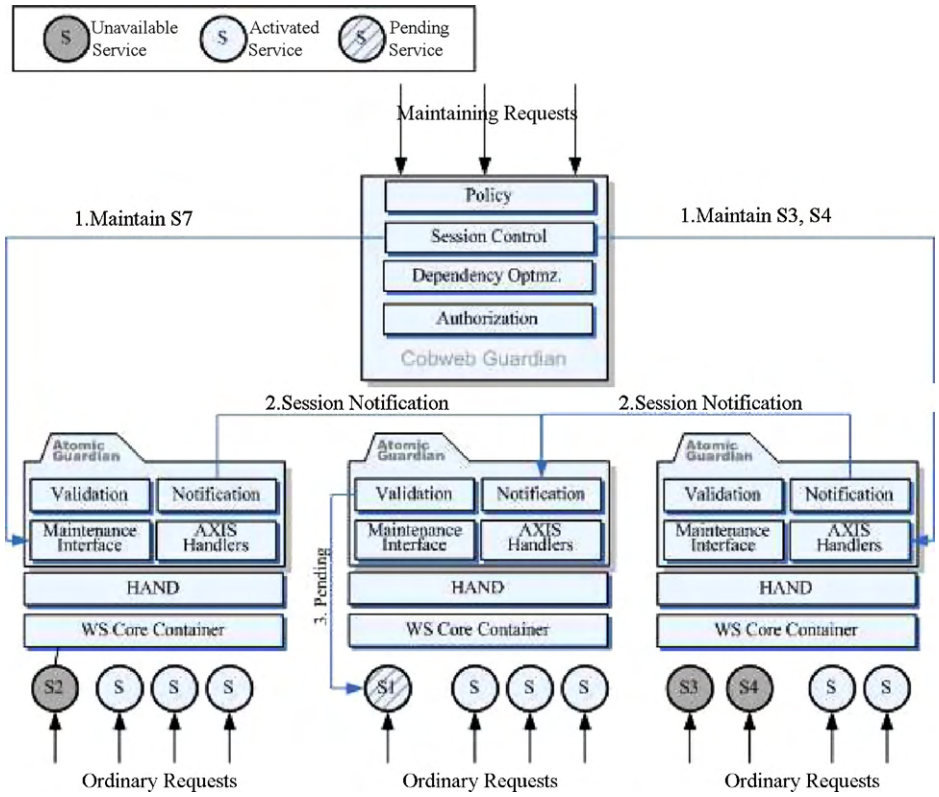


Fig. 2. Guardian software architecture.

(1) Session Control module is mainly in charge of controlling the progress of the maintenance. In addition to that, it also propagates the maintenance tasks to the target replica Atomic Guardian.
(2) Dependency Optimizer is the kernel module of CG. It parses administrator's command and then matches the command requirements to the existing dependency maps. The dependency relations in dependency map store at corresponding tables in MySQL database with version 5.0.33.
(3) Authorization module is designed for examining all of the maintenance requests to prevent the whole system from unpredictable dangers (e.g. the requests to deploy Trojan viruses).
(4) Policy module is designed for administrator to execute on demand maintenance.

Atomic Guardian, the actual maintenance executors, is implemented based on our former works (Qi et al., 2007). AG also consists of four parts:

(1) Notification module reports the maintenance status to depending services and the Cobweb Guardian. With this module, administrators can track the process of each maintenance task and detect possible failures at any time. The notifications would only be sent to peers which depend on the service under maintenance.
(2) Validation module is the sink of peer container's Notification module. AG will execute reloading or pending actions defined in policies to issue the corresponding notifications.
(3) Maintenance Interface accepts the requests from Session Control of CG and then executes the deploy, upgrade, or activate works by interacting with the management module of hosting container, e.g. HAND (Qi et al., 2007).
(4) Axis handler implements Apache Axis (version 1.2) handler interface *invoke()* and is designed to record the different invocation dependencies from the input and output message flow. Any recorded peer would be notified by the Notification module during the maintenance. It can efficiently help CG reducing the overhead.

By implementing the two layers, the deployment dependencies can be recorded and managed by CG, and the invocation dependencies can be correctly collect by the AG's handler. We can record, parse, analyze and then use these dependencies to reduce the maintenance time and improve the global availability.

In the next section, we will describe our solutions to optimize the maintenance procedure using the recorded dependencies in detail.

3.2. Environment dependency-aware maintenance in three granularities

By receiving and analyzing the maintenance requests, CG can adapt to different granularities (maintenance in service level, container level, and node level) according to related environment dependencies. Obviously the efficient reduction of maintenance granularity can help improving the efficiency of maintenance. Three-layer architecture (as shown in Fig. 3) is proposed in Cobweb Guardian to reduce the effects of dependency hierarchal.

Service level: It exists as a manager in the target hosting environment. The service-level maintenance means that all maintenance tasks are isolated for target service component. The reloading or pending operations are only executed as units of services. There-

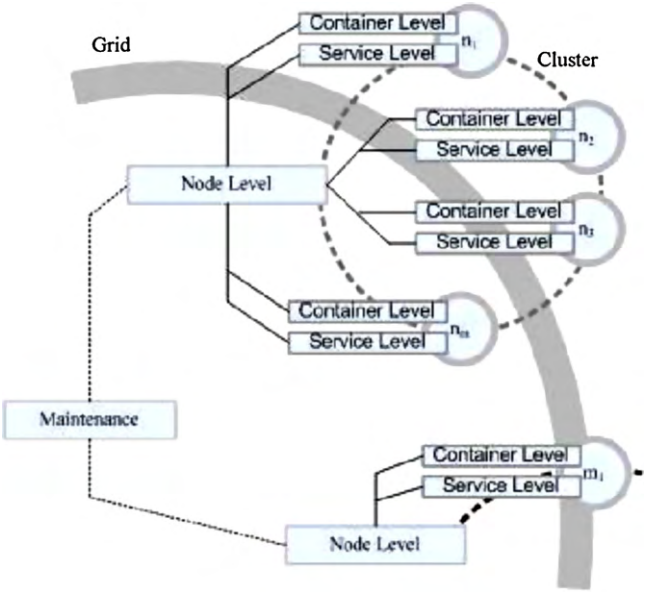


Fig. 3. Three-Layer architecture for maintenance.

fore other services in the same environment will not be affected.

Container level: It works in the hosting environment. Unlike service-level maintenance, the reloading or pending operations are for the whole container. That is, if a service needs to be maintained, the other services in the same container are also putted into the maintenance. This level is implemented widely in commonly used applications such as GT4, MySQL server, and HTTPD servers. Despite the granularity of this level is more complicated than service-level, it can save much time when maintaining a bunch of service components in a same container.

Node level: The reloading or pending operations in node level will affect a bunch of computing nodes from the global view. It adopts the maintenance policies among different containers. By communicating with the maintenance manager in service and container level separately (as shown in Fig. 3), it can promise that the optimized and safe maintenance solutions are adopted. For instance, we can avoid the effects from environment during the node level maintenance.

3.3. Deployment dependency-aware propagating maintenance

Similar to the actions of RPM (Mugler et al., 2005) manager in Linux system, the CG will check the dependency before the maintenance and generate the critical maintenance path to propagate the tasks. CG provides the session control mechanism to guarantee the execution of maintenance. The idea is from the parallel topological sorting algorithm (Kimelman and Zernik, 1993) and dynamical maintenance for k-connectivity Graph (Liang et al., 2001). CG propagates the maintenance tasks to the depending services first to guarantee the correct execution of the maintenance procedure. If the propagations are finished in k steps, the time cost of this solution can be calculated as:

$$t = \sum_{i}^{k} \max_{j}^{s(i)}(t^j) \tag{8}$$

The $s(i)$ in Formula (8) means the parallel maintenances for step i. And the availability is:

$$A = 1 - \frac{\sum_{i=0}^{k}(t_r^i + t_p^i)}{\sum_{i=0}^{n} t_i} \tag{9}$$

It denotes that the maintenance time is longer than the simple parallel solution in Section 2.3. However this solution can assure the correctness of maintenance for Grid administrators. Algorithm 1 demonstrates this procedure. Line 4 is to combine the possible maintenance tasks that deployment dependency related.

Algorithm 1. Deployment Dependency-aware Maintenance Algorithm.

input : $(S \times R)$and the requested maintenance op. op is the maintenance for service $s \in S$ (pre-)deployed on resource $r \in R$;
output: maintenance status

```
1  foreach (s, r) ∈ (S × R) do
       // invoke function GetDeployDependency to find out the deployment dependencies on
          resource r for service s.
2      (S_r, op) ← GetDeployDependency(s,r);
3      if S_r ≠ ∅ then
           // combine the deployment-depending tasks into task list to promise the
              correctness.
4          manstack ← manstack ⋃(S_r × {r}, op);
5      end
6  end
   // sort the depending tasks and execute concurrently.
7  status ← ParallelExecute(TopologySort(manstack));
8  return status;
```

3.4. Invocation dependency-aware grouping maintenance

To achieve higher availability during the maintenance, the different semantics of invocations are differentiated by AG.

(i) For AND-dependency. The maintenance for any service with this kind of dependency will lower the global availability since the unavailability of any component can be chained transmitted to the whole system. The best solution is to propagate all 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 proper policy (e.g. round robin, random, or load balanced policy), the global availability can be raised if the maintenance for composite services can be executed in groups. Formula (10) describes the improved availability by using this approach. The p_i ($0 < p_i < 1$) in Formula (10) means the possibility of dispatching requests to the maintenance containers. In addition, the t^0 means the maintenance time of the front end service component.

$$A = 1 - \frac{t_r^0 + t_p^0 + \sum_{i=1}^{k} p_i \cdot (t_r^i + t_p^i)}{\sum_{i=0}^{n} t_i} \tag{10}$$

However the maintenance time is also increased since we add the groups in serial (the bigger k in Formula (8)) for the maintenance. Algorithm 2 depicts the process of grouping the target resources. From line 7 to 9 we can find that CG will group OR and XOR-depending service components and maintain them earlier in the group (with higher priority).

3.5. Grouping maintenance with feedback

Although the grouping solution in Section 3.4 improves the availability by sacrificing the maintenance time, it also brings in some unpredictable factors to the system. For instance, some critical invocations will be rejected randomly. CG can provide the feedback notification interface for the Grid applications. By checking the status of remote service component, the Grid applications can bypass the requests to suitable maintenance service component. In this solution, the availability can be improved much while the cost is mainly from the maintenance of composite service component.

$$A = 1 - \frac{t_r^0 + t_p^0}{\sum_{i=0}^{n} t_i} \tag{11}$$

Formula (11) proved that the availability of this solution is better than serial, propagating, grouping maintenance solutions. Similar to Algorithm 2, Algorithm 3 groups the depending services first. The difference (line 10–12) is that the CG will query the feedback status from these remote service components in this solution. If one of them is unavailable, its priority value would be decreased and the its maintenance would be postponed.

Algorithm 2. Invocation dependency-aware grouping maintenance algorithm

```
input  : (S × R)and the requested maintenance op. op is the maintenance for service s ∈ S
         (pre-)deployed on resource r ∈ R;
output: maintenance status
1  foreach (s,r) ∈ (S × R) do
   |  // invoke function GetInvokeDependency to find out the deployment dependencies on
   |     resource r for service s.
2  |  (S_r,op) ← GetInvokeDependency(s,r);
3  |  if S_r ≠ ∅ then
4  |  |  switch TypeOf(S_r,s) do identify the type of invocations
5  |  |  |  case AND dependency
   |  |  |  |  ;                                          // do nothing for AND
6  |  |  |  end
7  |  |  |  case XOR or OR dependency
8  |  |  |  |  manstack (S_r).priority ++ ;               // group the services
9  |  |  |  end
10 |  |  end
11 |  end
12 end
   // sort the depending tasks and execute concurrently.
13 foreach each group do
14 |  status ← ParallelExecute(TopologySort(manstack),groupId);
15 |  if status = Failed then
16 |  |  break;
17 |  end
18 end
19 return status;
```

Algorithm 3. Invocation dependency-aware grouping maintenance with feedback algorithm.

```
input  : (S × R)and the requested maintenance op. op is the maintenance for service s ∈ S
         (pre-)deployed on resource r ∈ R;
output: maintenance status
1  foreach (s,r) ∈ (S × R) do
   |  // invoke function GetInvokeDependency to find out the deployment dependencies on
   |     resource r for service s.
2  |  (S_r,op) ← GetInvokeDependency(s,r);
3  |  if S_r ≠ ∅ then
4  |  |  switch TypeOf(S_r,s) do identify the type of invocations
5  |  |  |  case AND dependency
   |  |  |  |  ;                                          // do nothing for AND
6  |  |  |  end
7  |  |  |  case XOR or OR dependency
8  |  |  |  |  manstack (S_r).priority ++ ;               // group the services
9  |  |  |  |  foreach  s ∈ S_r do
10 |  |  |  |  |  if QueryFeedback(s,r) = unavailable then
11 |  |  |  |  |  |  manstack (s).priority −− ;          // postpone the maintenance
12 |  |  |  |  |  end
13 |  |  |  |  end
14 |  |  |  end
15 |  |  end
16 |  end
17 end
   // sort the depending tasks and execute concurrently.
18 foreach each group do
19 |  status ← ParallelExecute(TopologySort(manstack),groupId);
20 |  if status = Failed then
21 |  |  break;
22 |  end
23 end
24 return status;
```

4. Evaluations

We evaluated our implementation on Chinagrid test bed. Services in our experiments are deployed on Chinagrid Support Platform v2.0.1 (Wu et al., 2005).

Our evaluation has the following objectives:

- Demonstrate the improved availability by comparing the traditional maintenance approach with Cobweb Guardian.
- Compare the availability and throughput of different maintenance solutions of Cobweb Guardian when the maintenance happened in different levels.
- Demonstrate the effectiveness of dependency-aware mechanism with feedback under different maintenance granularity circumstances in improving service availability and throughput.

4.1. Test environment

Unless stated, experiments in this paper were conducted on two rack-mounted Linux clusters: One is with 16 1 GHz Pentium III nodes (each with 512 MB 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 1.3 GHz Itanium2 servers. The nodes inside the cluster are connected with a 100Mbps Ethernet switch. Each node runs Red hat 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 100 Mbps.

To support the correct execution of test case, we installed MPICH (version 1.2.4) and image processing toolkits on the target systems in advance.

Abbreviations: Here is a list of the abbreviations that we will use in the rest of this section:

- *NonD* denotes the normal maintenance solution in parallel like ProActive, or SmartFrog without dependency consideration;
- *SRL* is to maintain the services by calling shell scripts in serial;
- *CG-0* is the simple deployment dependency-aware propagating solution without optimization for invocation dependency;
- *CG-1* means invocation dependency-aware grouping maintenance solution;
- *CG-2* is the grouping maintenance solution with feedback;
- *REQ* in the diagrams denotes the fixed request rate for particular service components.

4.1.1. Application for benchmark

Our experiments are running on a typical executing system, called General Running Service(GRS, S1), implemented by CGSP. As shown in Fig. 4, the GRS includes six service components: authentication service (Auth, S2), information service (Info, S3), data management service (Data, S4), cache service (Cache, S5), collecting service (Collect, S6), and replication service (Replica, S7). The executing job issued in our experiment is a MPI-based image processing application (Haifang et al., 2005) that is deployed in our Chinagrid test bed. When a job request arrives, GRS parses it and then contacts with the Auth component to check the validity, the

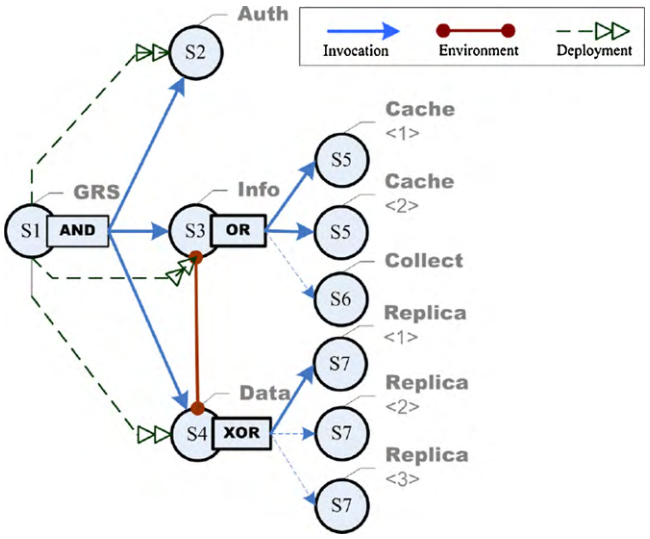


Fig. 4. Service dependencies in CGSP's execution system.

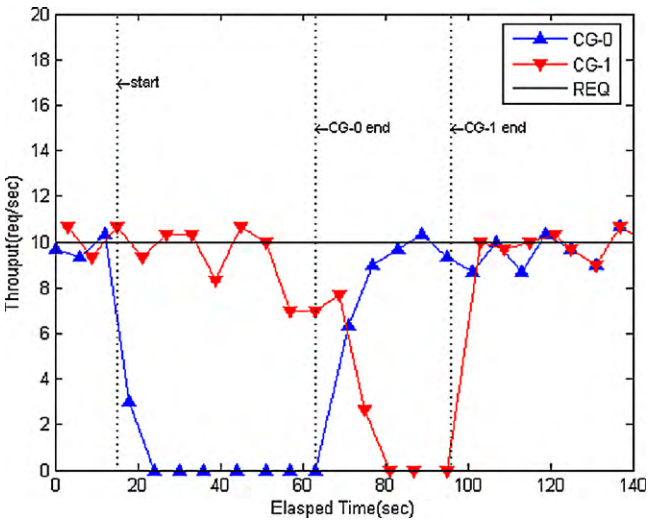


Fig. 6. Throughput of Info.

Info component to get job's executing information, and the Data component to fetch the staging data respectively. More particularly, the Info component will try to fetch the job information from the Cache first, invoke the Collect component to query in back-end database when failed. On the other hand, the Data component adopts a replication to store and load the staging data (usually a sample image). The staging data can have multiple partitions for Info and GRS. The detailed semantic about CGSP's execution system can be investigated in the literature (Wu et al., 2005). By injecting the maintenances procedure to these services lively, we can inspect the capability of Cobweb Guardian.

4.2. Deployment dependency-aware propagation

To investigate the efficiency of the propagation and the effects from deployment dependency, we employ the GRS service at 6 requests per second in this experiment. We run the experiment for 140 s. At second 15, we inject the maintenance procedure to the GRS (including upgrades for Auth, Info, Data service components, and itself) which then becomes unresponsive for a while after injection. We tested three solutions: NonD, SRL, and CG-0 in this experiment.

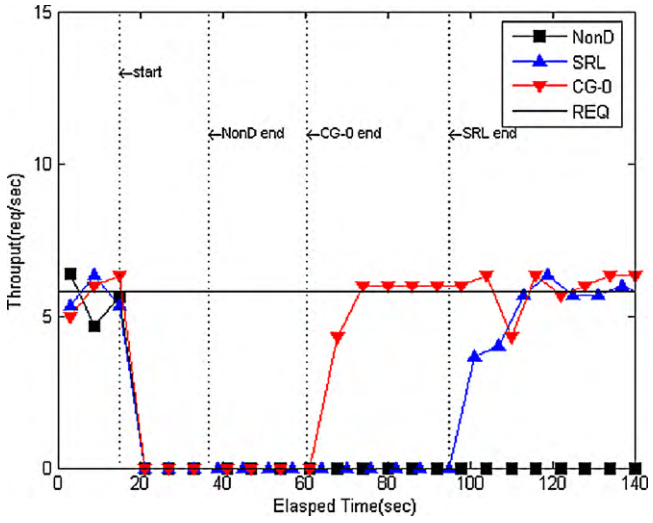


Fig. 5. The throughput of execution service during the maintenance.

Fig. 5 shows the throughput of three solutions during the 30-second watching period. The system works well before maintenance injection. When maintenances start at second 15, the throughputs of the system with different approaches were all falling down to zero. Although the NonD solution finished the maintenance earliest (shown in Fig. 5), the system could not work correctly any more. The maintenance for GRS component with this solution failed finally since it didn't consider about deployment dependencies between the services components. The SRL solution costs too much time to complete and affect system availability much compared to CG-0 although it successful finished maintenance. Against these two solutions, CG-0 realizes the highest efficiency. It lost 49.1% requests since the AND-depended service components are unavailable when any related service (i.e. Auth, Info, and Data) is under the maintenance.

4.3. Invocation dependency-aware grouping maintenance

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

4.3.1. OR-dependency

As shown in Fig. 6, we started maintenance solutions CG-0 and CG-1 respectively both at second 15. CG-0 propagated the maintenance tasks to the target containers that deployed Cache and Collect service components and then to the nodes with Info service. The whole procedure cost 62.97 s. However, from Table 1, we can find that the throughput during CG-0 execution is falling down near to zero. In addition to that, there are 57.9% lost requests in the watching period (defined in Section 2).

Compared to CG-0, the CG-1 optimized the solution for OR-dependency. The CG maintained the targeted containers in sequential with different priorities. The improvement of this solu-

Table 1
Comparison of experimental results.

Appr	Maintain	RespTime (ms)	Thrput	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

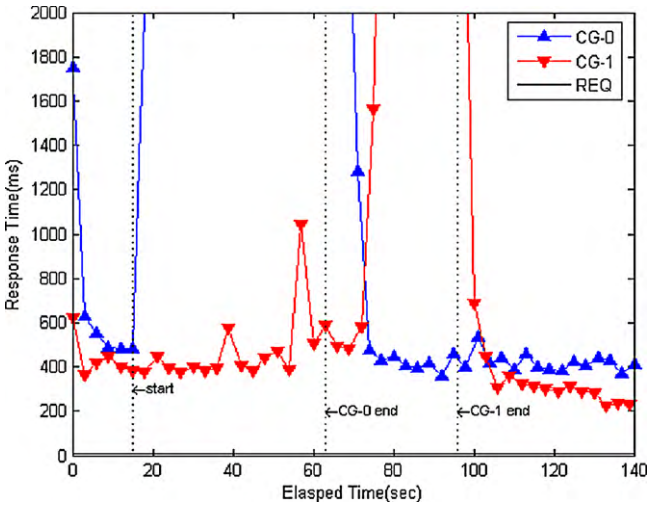


Fig. 7. Response time of Info.

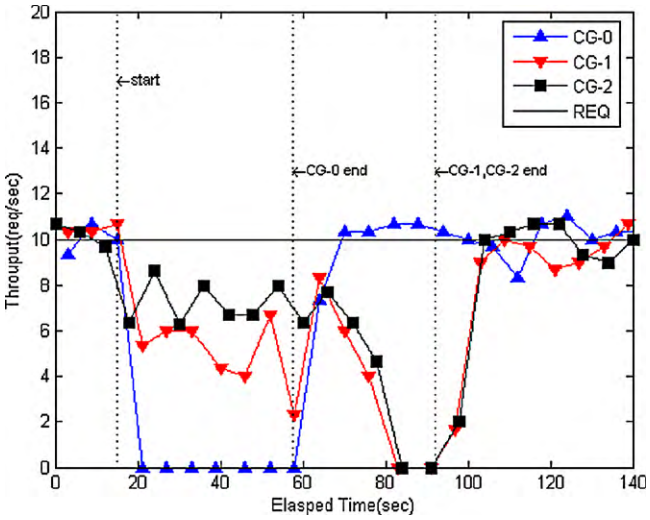


Fig. 8. The throughput of the virtual data center service during the maintenance.

tion 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 reduces from 57.9% to 25.6% with comparison to CG-0. 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 according to system execution semantics. From Fig. 7, we can identify that the throughput of CG-1 is falling down to about 7.3 from second 57 because the requests to Cache services were failed when they were under maintenance. Thus the requests were sent to the Collect service instead.

Fig. 7 denotes the response time for the procedures mentioned above. We can easily find that the average response time is increasing when Cache service began to maintain (at second 57). This increase is acceptable against solution CG-0. However, the improvement of CG-1 sacrificed the maintenance time, we had to take more 32.985 seconds to finish the maintenance job.

4.3.2. XOR-dependency

We repeated the experiment for Data service component (XOR-dependency). This time we introduced solution CG-2. This maintenance solution would feed back the maintenance status to the user-level applications. With these status feedback, some methods can be taken to avoid the accesses to the services that are under maintenance. To keep the consistency, we start the three maintenances at second 15 respectively too.

Fig. 8 describes the procedure: The CG-0 solution acted similarly like before. It blocked all requests to the Data service during the maintenance. Hence the throughput for CG-0 is lowest. However the CG-1 solution acted not well either. Although it improved the throughput to some extend (from 0 to 4.71), the loss rate is about 52.1% which is just a minor improvement to CG-0's 53.9%. The main reason is that the CG-1 cost more time to finish the maintenance than CG-0 as the Data service was always trying to deliver the requests to the Replica services during maintenance while these requests were failed definitely. Unlike CG-0 and CG-1, Solution CG-2 worked far better. It cost same time to finish maintenance like CG-0. However it gave better throughput (6.56), lower loss rate (25.9%), and better response time for normal requests (478 ms). Table 2 lists the average response time, throughput, and loss rate respectively for these three solutions in different stages (before and during) of maintenance. It clearly reflects the variations discussed above.

Table 2
Comparison of experimental results.

Appr	Maintain	RespTime (ms)	Thrput	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. Environment dependency-aware maintenance in different granularities

To demonstrate the enhancement on environment dependency, we executed the upgrade for Info service component meanwhile the Data service component was deployed in the same container.

Fig. 9 described the results. Fig. 9A denotes container-level maintenance and Fig. 9B is in service-level. As shown in this figure, in the container-level maintenance, the Data service was also unavailable when Info service is under upgrading. However, the maintenance for Info service in service level did not affect the access to Data service. In addition to that, the maintenance time in service-level (14.6 s) was also less than container-level (23.6 s). This result

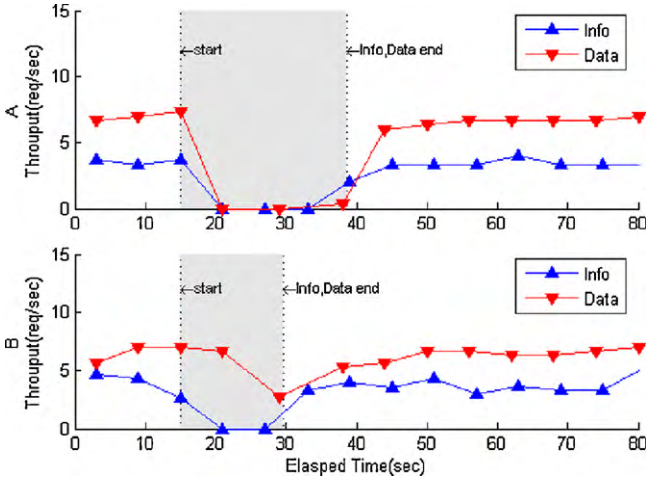


Fig. 9. Maintenance in different granularity for environment dependency.

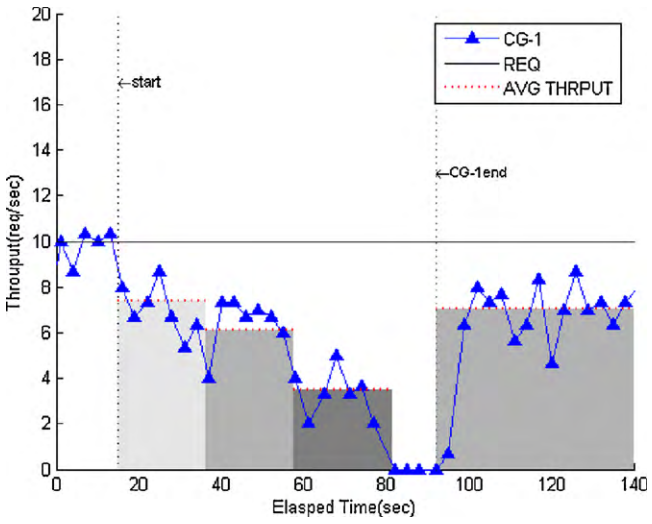


Fig. 10. Throughput variance when dynamically changing the services.

proved that the availabilities for Data and Info service components were enhanced.

4.5. Evaluation for dynamicity

According to the realistic statistics in Section 1, the biggest convenience for VO users is completing admin maintenances on infrastructure transparently and efficiently. In this experiment, we investigated the effectiveness for dynamical variation which is very common for Grid services. Besides the normal maintenance (transfer the upgrading packages, reload the hosting container, and so on) for Data and its Replica services, an un-deployment is executed for one of its replica.

The CG-1 solution is used in this experiment. Fig. 10 depicts the throughput's variation during maintenance. From the figure, we can find that after the whole maintenance the throughput was falling down to 2/3 of original throughput before the maintenance. This result proved that the Replica service was un-deployed successfully. In addition, the throughputs were lowering down in three stages during the maintenance procedure since the Cobweb Guardian blocked the requests to the service components which were under maintenance. Meanwhile, the loss rate during the maintenance is 25.7%. It is close to its minimal value.

5. Related works

The approach of maintenance software (or services) by exploring the dependencies among the software components has been widely discussed.

In the age of OO programming, Yau and Tsai (1987) proposed an approach using first-order logic for knowledge representation of software component interconnection information to facilitate the validity and integrity checking of the interconnection among software components during software development or modification. They used *Component Interconnection Graph* (CIG) to express the invocation dependencies. However the CIG cannot explain well the dependencies among different nodes when shift from object-oriented to service-oriented architecture.

Sangal et al. (2005) invented a dependency structure matrix to describe the dependencies among legacy software. It is efficient for exploring the software architecture from the view of software engineering. With a pity, they did not discuss the dynamic dependencies at runtime which are common in distributed and dynamical Grids. The dependency structure matrix is not convenient for dynamic deployment in Grid.

Service capsule proposed by Chu et al. (2005) is a new mechanism that supports automatic recognition of dependency states and pre-dependency management for thread-based services. Nevertheless capsule focus more on fault tolerant instead of the 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 literature (Qi et al., 2007; Talwar et al., 2005; Chu et al., 2005). 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 (Nagaraja et al., 2003).

In our former work (Qi et al., 2007), the two approaches (named service- and container-level) of dynamic deployment were proposed to enhance the availability of service infrastructure. The experiment result proved that the system availability can be improved to some extend by choosing the smaller granularity deployment (service level). However, whatever service- or container-level, the availability improvement is limited in infrastructure layer. It can't promise the global availability, especially when there were complex invocation dependencies among the services in different containers.

The *Configuration, Description, Deployment and Lifecycle Management* (CDDLM) specification (Configuration, 2005) proposed by OGF and the *Installable Unit Deployment Descriptor* (IUDD) (Installable) proposed by W3C are both the specifications to standardize the maintenance works for distributed software or services. But the two specifications do not promise the quality of maintenance tasks and the runtime availability during the maintenance. The SmartFrog (Smart frog project) project is a classic change-and-configure management tool for distributed services. However SmartFrog cannot promise the availability during the maintenance. OGSAConfig (Ogsaconfig; Smith and Anderson, 2004) aims at dynamically reconfiguring fabrics to enable each fabric to support a wider range of applications. CFengine (Cfengine) is designed as a reaction to the complexity and non-portability of shell scripting for Unix configuration management. CFengine and SmartFrog which based on the script and principles lacks the automaticity when configure the whole system while our Cobweb mechanism can automatically discover the dependencies to form a dependency map and then report to related components to avoid unnecessary lost.

Due to the complexity of networking environment, the P2P-based computing infrastructure are much harder to maintain. Tamimi (2007) proposed an automated peer-to-peer security-update services. To protect the network against malicious computers that may join in to spread infected files, authors address more on security instead of the dynamicity of infrastructure. By combining semantic technologies, Zhuge et al. (2005) investigated the deployment of a scalable distributed trie index for broadcast queries on key strings, propose a decentralized load balancing method for solving the problem of uneven load distribution incurred by heterogeneity of loads and node capacities. Different from scalability, our solution concentrate much on dynamicity of highly shared Grid infrastructure.

Talwar et al. (2005) compared manual, script-, language-, and model-based deployment solutions in terms of scale, complexity, expressiveness, and barriers for distributed services. Despite the dependency problem was discussed, the effects to the availability of the whole system during deployment for system was not discussed in that paper.

Nagaraja et al. (2004) discussed the operator mistakes in Internet services. The paper demonstrates how to detect the operator mistakes via the creation of a validation environment that is an extension of the online system, where components can be validated

using real workloads before they are migrated into the running service. However this solution is to detect and avoid the maintenance mistakes, it cannot optimize the runtime availability.

6. Conclusion and future works

In this paper, we propose the Cobweb Guardian which is a dependency-aware maintenance architecture for service-oriented Grids. By investigating effects from different dependencies at runtime (including invocation-, deployment-, and environment-dependency), the Cobweb Guardian can automatically generate the optimized solutions for the maintenance in distributed Grids. The evaluation results demonstrate the effectiveness of the Cobweb Guardian of improving the availability and throughput during the maintenance.

The further works include the investigation on fault tolerance for distributed maintenance since the failed maintenance affects the availability of the whole system. In addition to that, the challenge from the Quality of Service at runtime is another important expanding point of our CG system.

Acknowledgments

This work is supported by ChinaGrid project, China Next Generation Internet Project under Grant CNGI2008-109, National High-Tech R&D Plan of China under Grant 2006AA01A115, Program for New Century Excellent Talents in University under Grant NCET-07-0334.

References

Beowulf introduction and overview. http://www.beowulf.org.
Cfengine. http://www.cfengine.org/.
Ogsaconfig. http://groups.inf.ed.ac.uk/ogsaconfig/.
Oscar: A packaged cluster software stack for high performance computing. http://www.openclustergroup.org/.
Proactive project. http://proactive.inria.fr/.
Smart frog project. http://www.hpl.hp.com/research/smartfrog/.
Configuration, deployment description language and management. Technical report, Open Grid Forum, 2005.
Installable unit deployment descriptor specification. Technical report, W3C, 2005.
Chu, L., Shen, K., Tang, H., Yang, T., Zhou, J., 2005. Dependency isolation for thread-based multi-tier internet services. In: INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE, vol. 2, pp. 796–806.
Foster, I., 2006. Globus toolkit version 4: Software for service-oriented systems. Journal of Computer Science and Technology 21, 513–520.
Foster, I., Kesselman, C., Tuecke, S., 2001. The anatomy of the grid: enabling scalable virtual organizations. International Journal of High Performance Computing Applications 15 (3), 200–222, 474ZP Times Cited:312 Cited References Count:63.
Hai Jin, S.W.J.D., Qi, L., Luo, Y., 2007. Dependency-aware maintenance for dynamic service grid. In: Proceedings of the 36th International Conference on Parallel Processing (ICPP07). IEEE Computer Society, pp. 64–72.
Haifang, Z., Xuejun, Y., Hengzhu, L., Yu, T., 2005. First evaluation of parallel methods of automatic global image registration based on wavelets. In: International Conference on Parallel Processing, 2005. ICPP 2005., pp. 129–136.
Karonis, N.T., Toonen, B., Foster, I., 2003. Mpich-g2: a grid-enabled implementation of the message passing interface. Journal of Parallel and Distributed Computing 63 (5), 551–563.
Kimelman, D., Zernik, D., 1993. On-the-fly: Topological sort a basis for interactive debugging and live visulization of parallel programs. In: Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, pp. 12–20.
Liang, W., Brend, R., Shen, H., Aug 2001. Fully dynamic maintenance of k-connectivity in parallel. IEEE Transactions on Parallel and Distributed Systems 12 (8), 846–864.
Mugler, J., Naughton, T., Scott, S.L., 2005. Oscar meta-package system. In: High Performance Computing Systems and Applications, 2005. In: 19th International Symposium on HPCS 2005, pp. 353–360.
Nagaraja, K., Li, X., Zhang, B., Bianchini, R., 2003. Using fault injection and modeling to evaluate the performability of cluster-based services. In: Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems.
Nagaraja, K., Oliveria, F., Bianchini, R., et al., Dec 2004. Understanding and dealing with operator mistakes in internet services. In: Proceedings of 6th USENIX Symposium on Operation Systems Design and Implementation (OSDI'04), San Francisco, CA, pp. 61–76.
Qi, L., Jin, H., Foster, I., Gawor, J., Feb 2007. Hand: Highly available dynamic deployment infrastructure for globus toolkit 4. In: Proceedings of the 15th Euromicro Conference on Parallel, Distributed and Network-based Processing, Naples, Italy, pp. 155–162.
Sangal, N., Jordan, E., Sinha, V., et al., 2005. Using dependency models to manage complex software architecture. In: Proceedings of the OOPSLA'05, pp. 167–176.
Smith, E., Anderson, P., 2004. Dynamic reconfiguration for grid fabrics. In: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, pp. 86–93.
Talwar, V., Qinyi, W., Pu, C., Wenchang, Y., Gueyoung, J., Milojicic, D., 2005. Comparison of approaches to service deployment. In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, 2005. ICDCS 2005, pp. 543–552.
Tamimi, Z.M., 2007. Automated peer-to-peer security-update propagation network. In: ICCOMP'07: Proceedings of the 11th WSEAS International Conference on Computers. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, pp. 557–564.
Wu, Y., Wu, S., Yu, H., Hu, C., October 2005. Cgsp: An extensible and reconfigurable grid framework. In: Proceedings of the 6th International Workshop on Advanced Parallel Processing Technologies, Hong Kong, China, pp. 292–300.
Xu, Z., Li, W., Zha, L., 2004. Vega: a computer systems approach to grid computing. Journal of Grid Computing 2 (2), 109–120.
Yau, S., Tsai, J., 1987. Knowledge representation of software component interconnection information for large-scale software modifications. IEEE Transactions on Software Engineering SE-13 (3), 342–355.
Zhuge, H., Sun, X., Liu, J., Yao, E., Chen, X., Dec 2005. A scalable p2p platform for the knowledge grid. IEEE Transactions on Knowledge and Data Engineering 17 (12), 1721–1736.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at the Huazhong University of Science and Technology (HUST) in China. He is now Dean of the School of Computer Science and Technology at HUST. Jin received his PhD in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of National 973 Basic Research Program Project of Virtualization Technology of Computing System. Jin is a senior member of the IEEE and a member of the ACM. Jin is the member of Grid Forum Steering Group (GFSG). He has co-authored 15 books and published over 400 research papers. His research interests include computer architecture, virtualization technology, cluster computing and grid computing, peer-to-peer computing, network storage, and network security. Jin is the steering committee chair of International Conference on Grid and Pervasive Computing (GPC), Asia-Pacific Services Computing Conference (APSCC), International Conference on Frontier of Computer Science and Technology (FCST), and Annual ChinaGrid Conference. Jin is a member of the steering committee of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid), the IFIP International Conference on Network and Parallel Computing (NPC), and the International Conference on Grid and Cooperative Computing (GCC), International Conference on Autonomic and Trusted Computing (ATC), International Conference on Ubiquitous Intelligence and Computing (UIC).

Yaqin Luo is a PhD candidate in computer science at Huazhong University of Science and Technology, China. She received her B.E. degree in computer science at Huazhong University of Science and Technology, China. Her current research interests include grid computing and cloud computing, failure-aware resource management in distributed computing and virtualization technology. She received the excellent thesis award of Hubei province, China, in 2004.

Li Qi is a software engineer and project manager in China Development Bank. He received his B.E., M.S. and PhD degrees in computer science at Huazhong University of Science and Technology, China. His current research interests are grid computing, cloud computing, and financial IT.

Jie Dai is a PhD candidate in Hong Kong University of Science and Technology, China. He received his B.E. and M.S. degrees in computer science at Huazhong University of Science and Technology. His research interests include grid computing and mobile computing.

Song Wu is a professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He received his PhD from HUST in 2003. He is now the Vice Head of Computer Engineering Department at HUST. He is also served as the Vice Director of Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) of HUST now. His current research interests include grid/cloud computing and virtualization technology. He has been involved in the most important Grid projects (CNGrid and ChinaGrid) of China for more than 5 years.