The Session Based Fault Tolerance Algorithm of Platform EGO Web Service Gateway

Xiaohui Wei, Yuan Luo, Jishan Gao, Xiaolei Ding
College of Computer Science and Technology, Jilin University, China
{weixh@jlu.edu.cn, pp.jordan@email.jlu.edu.cn, gjs0064114@126.com, dxlxiaolei@163.com}

Abstract

Although grid computing has adopted Web services technology to deal with platforms heterogeneity and to enhance service and application interoperability, it is still a challenge to build web service applications with high reliability and availability to meet the requirements of grid communities. The paper discusses the design of Platform EGO WSG with high reliability. To support a huge user base and reduce the response time, WSGs work in cluster model and the loads are dynamic balanced among them. Besides, a lightweight notification mechanism is implemented to provide better interoperability between WSG and WSCs. Moreover, we designed a session-based a-synchronized recovery algorithm to achieve WSG fault tolerance, which has short freezing time and is able to isolate the recovery process for each WSC. This approach can rebuild the service sessions and the notification mechanism after restart, to handle Notification failure, and WSG failure report, etc.

Keywords

grid, web service gateway, fault tolerance, session, load balance

1. INTRODUCTION

From OGSI to WSRF, grid computing has gradually adopted Web services and SOA technologies to solve the resource sharing problems in heterogeneous environments of science, engineer and commerce[1]. Unlike traditional cluster computing environments such as LSF, PBS, and SGE etc, Platform Enterprise Grid Orchestrator™ (EGO) is a SOA based grid platform newly released by Platform Computing Inc. to manage the shared resources across geographically dispersed sites for diverse enterprise applications, services and workloads. Platform EGO Web Service Gateway(WSG), is a grid middleware to enable the applications, called web service clients(WSC), to access Platform EGO services as web services. However, it is a challenge to realize web service based grid services with high performance, reliability and availability to meet the requirements of grid communities. As grid computing becomes widely adopted, there is a fresh need for web service technologies to combine with recovery-based techniques and parallel processing technologies to achieve fault-tolerance and high performance.

1.1 Related Works

Although many works have been done in the field of distributed system recovery, the research on web service fault-tolerance is very new in this area. Currently there are no standard specifications dealing with fault tolerance in web services.

Normally a web server does not maintain the active connections with its clients, which is called stateless. Hence, in many cases, people just use very simple protocol to handle the web service crashes. A service monitor mechanism would be used to detect the service fault and the future requests from clients will be re-directed to redundant servers. For example, paper [2,3] deliver fault tolerance on web services based on the passive replication approach and implement basic fault detection mechanisms on primary server. Paper [4] proposes a general architecture to realize fault-tolerance web services, which have components responsible for calling concurrently the service replicas, wait for processing, analyze the responses processed, and

return them to the client. Moreover, paper [4] supports the use of the active replication technique in order to obtain fault tolerance in service oriented architectures.

Paper [3] is also capable of tolerating for requests being processed at the time of server failure. However, its implementation need modifications to the Linux kernel and to the Apache web server. While paper [5] presents an implementation of a fault tolerant TCP that allows a fault tolerance server to keep its TCP connections open until either it recover the TCP connection or fail to backup. Working with rollback recover, the failure and recovery of the server process are completely transparent to client processes connected with it via TCP.

In Platform EGO, the pattern of Web Service Gateway is used to trap and map service requestors to its target services. To support a huge grid user community, Platform Ego WSG can be deployed in cluster model in that a bunch of WSGs work concurrently with dynamic load balancing to provide a much higher performance. As the numbers of WSGs could be large and their locations are not fixed, it is not practical to setup a backup for each WSG， due to the performance overhead[6]. Hence, in this paper, we use rollback recovery to realize a lightweight fault-tolerance mechanism for WSGs. It works well with the WSG cluster model to be able to provide both high reliability and high performance to end users.

Rollback recovery achieves fault tolerance by saving the recovery information (called checkpoints) of processes periodically in stable storage, which has many flavors. It can be transparently to users via supported by OS kernel, like on Cray or SGI, or implemented as a library, like Condor[7]. It can also be embedded in applications to let users decide when and what to save on stable storage, which is called application level rollback recovery.

In the paper, the application level rollback recovery is realized in WSG. Since the most critical information in a WSG is the active sessions between WSG and its WSCs, and the sessions between WSG and EGO internal services, we designed a session based a-synchronized recovery algorithm which has short freezing time and is able to isolate the recoveries of different WSCs. Moreover, the load balancing algorithm for WSG cluster is also based on sessions, which is consistent with the recovery algorithm. When a WSG is down, the system can either restart a new instance if there is an

available host, or select another WSG to take over the failed WSG's workload.

1.2 Paper Organization

The rest of this paper is as follows. In Section 2 we introduce the overview of EGO platform. Section 3 presents the EGO WSG, including WSG architecture, WSG session, WSG security, etc. Section 4 gives out the WSG fault tolerance approach and recovery algorithm based on Reliable Notifications, WSG failure report, etc. In section 5, we make the conclusion.

2. EGO OVERVIEW

To discuss Platform EGO is out of the scope of the paper. However, in order to understand WSG's functionality, we will give a brief introduction to Platform EGO first. Platform EGO is a SOA based grid platform to offer a single, cohesive management environment that centrally allocates the shared resources across geographically dispersed sites for diverse enterprise applications, services and workloads. It allows developers, administrators, and users to treat a collection of distributed software and hardware resources on a shared computing infrastructure (cluster) as parts of a single virtual computer. Platform EGO uses Information, Allocation and Execution as key concepts in its Enterprise Grid Architecture. While many technologies effectively deal individually with Information and Execution activities associated with resource management, none take a comprehensive approach to the Allocation component. To accomplish this, Platform EGO uses a single common agent on each server to orchestrate the sharing of enterprise resources across application and organizational domains. Figure 1 illustrates Platform EGO as the foundation for a grid platform. The traditional computing resources, like hosts and clusters, are virtualized by a bunch of loose-coupled services, such as resource allocation service, execution service, security service, etc.

In a traditional cluster, like LSF or SGE, the users submit their jobs to the cluster. Then, the cluster will allocate resources and execute the jobs. A

couple of EGO services provide the similar functionality but with better flexibility and extensibility. For example, in EGO, a user may first ask for resources from Allocation Service. Once the resources are allocated, Allocation Service will send a notification to the user. Then the user can ask for Execution Service to execute the task on allocated resources. If the resources allocated to a user are reclaimed by the system, or the status of a task change, the user will also get the notifications from EGO services.

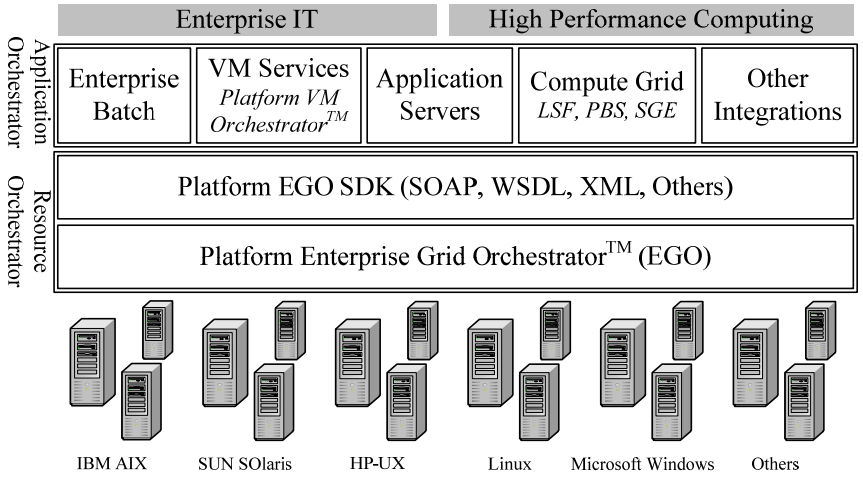

Figure 1. Platform EGO as the foundation for a grid platform

3. EGO WEB SERVICE GATEWAY

EGO Web Service Gateway (WSG) that provides a standards-based web services interface for web service clients (applications) to contact Platform EGO.

3.1 EGO WSG Architecture

WSG is a special EGO service to enable the users to access EGO via web service interface, and itself is also under the control of Service Director. The WSG is able to (a) transfer a WSC's request to the proper EGO services; (b) send the notifications from EGO services back to WSCs; (c) support role-based access control; (d) has no effect on the WSCs after a restart, (e)

provide EGO Platform with high performance. In the paper we introduced the WSG Cluster model to enable multiple WSGs to work together to provide higher performance and scalability. Figure 2 shows the WSGs Cluster Architecture.

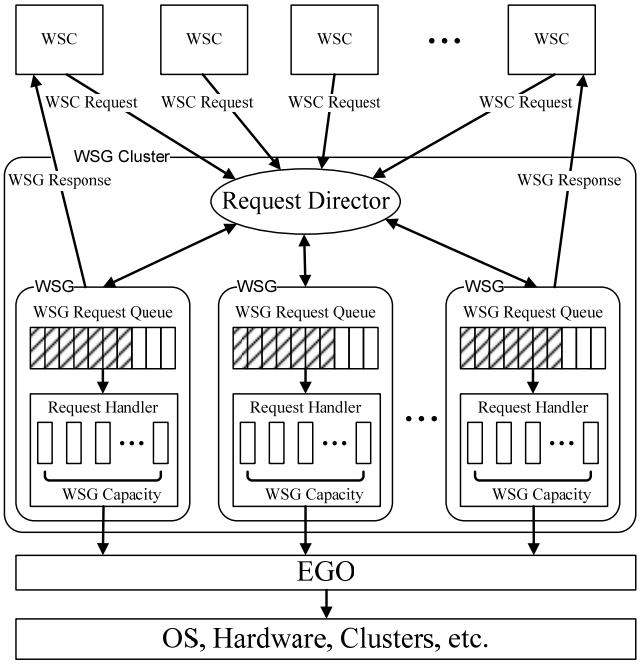

Figure 2. WSGs Cluster Architecture

In this model, multiple WSGs are working in parallel to handle WSC requests, and balance the load among them to further improve the scalability and performance, which is similar to WebSphere's Web Service Gateway Cluster(WSGC). A new component, Request Director is introduced to distribute the load among WSGs. Request Director contains two sub components: Load Balancer and Load Info Manager. The Distributor of each WSG will periodically send its load info L to Load Info Manager. If the load of a WSG is too high, it even can send an *unavailable* message to Load Info Manager to block Balancer to distribute more loads to it.

Load Balancer is a mediator between WSCs and WSGs, and its load distributing policy is also session based. At the beginning of a new client session, WSC will query Load Balancer to get an available WSG. Then, Load Balancer will check Load Info Manager and return a proper WSG's URL to WSC. After that, the WSC will contact the WSG directly during the

session. If there is no WSG available, Load Balancer may start a new WSG. Hence, the WSG Cluster size is dynamic adjusted due to the real load.

WSG uses the thread pool pattern to process WSC requests in parallel. There is a Request Queue to hold the requests from WSCs, and there is a bunch of working threads, called Request Handlers, to handle the requests in Request Queue concurrently. Distributor is responsible for dispatching the requests to the Handlers. The handlers access the proper EGO services on behalf of WSCs and send back the results to WSCs. Figure 3 shows the WSGs workloads.

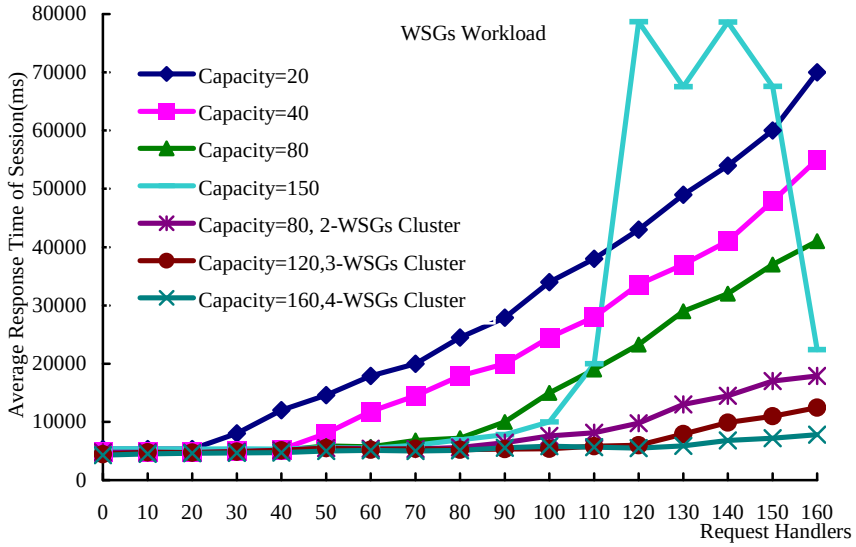

Figure 3 WSGs Workload

Each Handler processes the WSC's requests by transforming them into a series of accesses to EGO services using internal APIs. Some EGO services require the clients to maintain the session context, and the related communications must use the same session Handle. However, WSCs have no knowledge about the EGO services. Hence, WSG implements a Session Manager to manage the EGO service sessions for WSCs. While accessing EGO services, Session Manager will decide whether to use an existing session or to create a new one. If a session will not be used any more, it will be closed by Session Manager immediately.

The WSG supports notifications by following WS-Notification

specification[9]. Notification Manager integrates the functionalities of Notification Broker and Subscription Manager. EGO services work as Notification Publishers, while WSCs are both Notification Subscribers and Notification Consumers. The WSCs who want to receive notifications should register themselves to WSG Notification Manager first. Then, Notification Manager will listen to the active sessions maintained by Session Manager, collect the notifications from EGO services, and deliver them to registered WSCs respectively.

Due to the Notification Manager and Session Manager, WSG could not be designed as a stateless component. After restart, WSG must rebuild all the live sessions with EGO services, and recover the notification context for all registered WSCs. Hence, Data Manager is designed to save/restore the necessary running contexts related to notifications and sessions. We will discuss WSG's recovery algorithm in Section 4.4.

3.2 WSG Sessions

Session is an important concept in WSG, as it relates to WSG's recovery, notification, and performance tuning etc. WSG has two kinds of sessions. One is the sessions between WSCs and WSG, which are called client sessions. The other is the sessions maintained by WSG and EGO services, which are called service sessions. After a client session is created, multiple service sessions would be setup by WSG and specific EGO services to perform the requests from the client session. Once a client session is closed by the WSC, all the corresponding service sessions will be closed immediately by WSG. The following example will show how WSC run a task on Platform EGO via WSG.

Commonly, a client session contains multiple operations, and Figure 4 gives out an example. First, the WSC starts the client session with WSG by providing the role name and password (step 1). WSG will authenticate the role name and password via accessing EGO security service (step 2), and return a session credential to WSC on success (step 3). The WSC should ship the credential in its subsequent requests to identify itself during the session. In step 4, WSC sends the resource requirement for the job execution to WSG and waits for resource allocation notifications. Then, WSG will

create a service session with EGO Allocation Service and send the resource allocation requests to it (step 5-1). After the required resources are allocated successfully, a notification will be returned by Allocation Service to WSG (step 5-2). Once received the resource allocation notification forwarded by WSG (step 6), the WSC will start the job execution request (step 7). After that, WSG will create another service session with Execution Service to start the job (step 8-1). During the job running, the notifications will be sent to WSC once the job status changes (step 8-1). After the job finish, WSG will send a job finish notification to WSC (step 9). Then, WSC will close its session with WSG (step 10), and WSG will cancel the credential and close all the relevant service sessions.

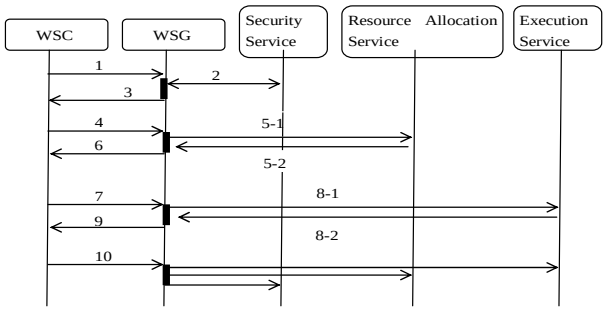

Figure 4. A Client Session

As there are dependencies between some operations in a session, such operations have to be executed in proper order. For example, the job execution operation must be issued after its resource allocation operation. In real life, the operations from different client sessions (WSCs) will compete for EGO resources. If an operation of a session is delayed, all its consequent operations are delayed too.

3.3 WSG Security

Platform EGO provides the role-based access control for WSCs. A user could be mapped to multiple roles inside of EGO. Before accessing any EGO service, a WSC must authenticate itself to EGO. WS-Security's Username Token is used between WSG and WSCs to support such authentication. In WS-Security's Username Token profile 1.0[8], a user name token consists of four elements as below:

UsernameToken: (Username, Password, Nonce, Created)

Nonce element is a random value created by the sender which is used as an effective countermeasure against replay attacks. The *Created* element specifies a timestamp which is used to indicate the creation time. The *Username* and *Password* pair is used to authenticate the WSC's identity. Inside Platform EGO, there is a table to map the usernames to role names. Besides normal users, there are some special users are defined by Platform EGO, such as EGO_admin, which has privileges to re-start the EGO system or change the system configuration.

As the internet is an unsafe communication channel, even WSG supports SSL to protect the SOAP message from being intercepted by eavesdroppers, it is still not recommended to use Username/Password frequently. Hence, we enhanced the Ws-Security Username Token profile to support credential based authentication. A credential is an encrypted string token that consists of the user information and a time stamp. Normally, a WSC gets its credential from WSG by providing the username and password when it starts a new client session with WSG. After that, WSC is able to use Credential instead of Username/Password to authenticate itself to WSG. Since each credential has a life time like a Kerberos ticket, it needs to be renewed before it expires during a session. With the above enhancement, a username token is extended as the following:

UsernameToken: ((Username, Password)|(Credential),Nonce, Created)

The above enhancement does not only improve the protection for sensitive information but also provide more flexibility. For example, the system admin does not need be mapped to privilege roles for all his operations. He can start a session as a normal user and get the credential. In most cases, he just uses the normal user's credential to perform the tasks, and the system admin's username/password is only used when necessary.

4. WSG FAULT TOLERANCE

The communications among WSCs, WSG and EGO services are not simple request-response model. Notifications play very important roles so that the WSCs are usually designed as event-driven applications. Therefore, WSG must have its own fault tolerant approaches, such as rebuild the service

sessions and the notification mechanism after restart, to handle the failures. Otherwise, it could cause deadlocks or WSG crashes.

4.1 Reliable Notifications

A deadlock may be caused when a WSG have already sent the allocation notification to the WSC but the WSC do not receive any notification. In this scenario, the allocated resource will not be released unless the WSC calls the WSG to do so. But the WSC has not even been notified that the required resources have been allocated.

In this case, A WSG has the responsibility to notify its WSCs with any status change, such as Allocation Notification, et al. In this scenario, the WSG is a A WSC, in another hand, have the responsibility to inform the WSG when the notifications of resources allocation have been received.

Firstly, the WSG disseminates information as a Notifications Publisher by sending one-way messages to the WSC as a Notification subscriber that are registered to receive it. Secondly, the WSC plays a role of Notification Publisher to send the notification back to the WSG while WSG is a Notification Subscriber. If a WSG doesn't hear from a WSC about the notification for a while, a copy of this notification will be sent again. But if the WSG crashes in the middle of this resend procedure, the deadlock will show up again. In section 4.3, we will introduce Session Recovery to prevent this deadlock caused by WSG failure.

4.2 WSG Failure Report

Previously, The WSG Director has no instrument to be informed if a WSG is disabled to connect to its Clients. Here we report a proper method for the WSC client to inform the WSGs Director about WSG failures. When a WSC has a problem when connecting to a WSG, it reports an error code to WSGs Director with this failure. With this report in hand, the WSGs Director evaluates the failure and restart the WSG if needed.

An API to this method is implemented on the Client side of WSG. The Failure Report method consists of 3 parameters: 1) Client Session ID; 2)

WSG handle; 3) Error Code.

Using this method, the WSCs group can partially be a WSGs status monitor, and significantly increase the reliability of the WSG system.

4.3 Sessions Recovery

In Figure 4, WSG crashes and restarts at some time between step 5 and step 6. After restart, WSG cannot build the service sessions for WSC by itself as it doesn't know either the session's credential or WSC's user-name/password. However, the WSC will not send any further request unless it receives the resource allocation notification. Unfortunately, due to the notification failure caused by service sessions lost, a deadlock comes up. Moreover, the resources allocated for the WSC will not be used or released. In this scenario, we recover the client session and service session when restart the WSGs.

Table 1. Recovery Table

Client Session ID	**Flag**	**Notification End point**	**Service Name List**	**Notification Not Received**		
ClientSession1	VALID	http://wsc1.jlu.edu.cn/8800	EGO service1			
			EGO service2	Note1	Note2	
ClientSession2	VALID	http://wsc2.jlu.edu.cn/8801	EGO service1	Note1		
			EGO service2	Note2	Note3	Note4

As WSG decides when and what should be saved, only necessary information is saved for the recovery, which is one of the advantages of application level checkpointing. The below table, Recovery Table, will be maintained by WSG Data Manager in the disk. The table records all the active client sessions identified by session ID, WSCs' endpoint to receive notifications, the EGO services that a WSC is accessing, and Allocation Notifications which are not received and reported back from WSCs. After restart, the recovery algorithm will recover all the existing client and service sessions. When the sessions between WSCs and WSGs rebuilded, the WSGs will send again the Allocation Notifications exist in the table, to the WSCs. The Allocation Notifications will be removed from the recovery table when

the WSG get the reply from the WSC. If all the Notifications have been received by WSCs, the Column “Notification Not Received” in the recovery table will be blank.

4.4 Recovery Algorithm

As the WSCs are designed by the different users, they could have different working models. Hence, the WSG’s recovery algorithm must consider all the possible behaviors of WSCs. And we should isolate the recovery procedures for different WSCs so that they will not affect each other. The checkpointing/restart algorithms are embedded in WSG’s source code.

The recovery algorithm consists of six parts.

Algorithm Part 1: After restart, WSG will rebuild the Recovery Table and start a dedicated recovery thread to do the recovery as below.

{

*Build the **Recovery Table** from disk;*

*Mark all the client sessions in the Table as **INVALID**;*

Create the recovery thread dedicated for recovery, then:

*The **Main thread** will execute algorithm part 2;*

*The dedicated **recovery thread** will execute algorithm part 3;*

}

Algorithm Part 2: At normal runtime, all the active client sessions in the Recover Table are marked as VALID, and WSG will execute this part of Algorithm.

While (true){

Get a Request from WSC;

*If (the Request is from an **INVALID client session**) Then {*

// this “ branch” will only be executed after WSG restart.

Get session credential from the request’s SOAP message header;

Rebuild all the service sessions with the services in the Service Name List;

*Mark the client session as **VALID**;*

};
Handle the Requests;
*If (the Request is **created a new client session**) Then {*
Append a new item in the recovery table;
Fill the WSC's Notification Endpoint and the client session ID;
*Mark the new client session as **VALID**;*
};
*If (the Request is **closed a client session**) Then {*
Remove the item from the recovery table for the client session;
};
*If (the Request is **created a new service session**) Then {*
Execute algorithm part 4;
Append the service name into the Service Name List for the proper client session;
}
*If (the Request is **closed a service session**) Then {*
Remove the service name from the Service Name List for the proper client session;
};
Get a message from WSC;
*If (the message is **Notification Received**) Then {*
*Remove the Notification from the **Recovery Table;***
}
*If (the contents of the **Recovery Table** changes) Then {*
Save the Recovery Table into the disk;
}
}

Algorithm Part 3: The recovery thread will execute the following algorithm.

Do {
*Send WSG **Restart notification** to all the WSCs with an client session marked as **INVLAID** in Recovery Table;*
//WSCs will execute algorithm part 5 while get the notification
While (there are replies from WSCs) {

// handle WSCs' replies for WSG Restart notification
Get the reply from next WSC;
*If(the WSC's client session is marked **INVALID**) Then {*
Get the session credential from the reply's SOAP message header;
Rebuild all the service sessions with the services in the Service Name List;
*Mark the client session from this WSC as **VALID**;*
} Else { // ignore the redundant replies.
Discard the reply;
}
}
*} Until (all the client sessions are **VALID**)*
Exit the thread;

Algorithm Part 4: This part of algorithm is executed by Request Director to return a lightest workload WSG to the WSC, or restart a failed WSG.

{
*If(**create a new service session** request) Then{*
If(No WSG avaiable) Then {
Start a new WSG and return the WSG URL to WSC;
}else{
Return a lightest workload WSG to WSC;
}
}
*If(**WSG Recovery Request**) Then{*
Go to Algorithm part 3; //Restart the failed WSG;
}
}

Algorithm Part 5: This part of the algorithm is executed by the WSCs notification handlers, to report back any information.

{
*If (it is a **WSG Restart notification**) Then {*
Put client session's credential in the reply's SOAP message header;

Send the reply to WSG;
}
*If(it is a **Resource Allocation Notification**) Then{*
Send a message to WSG that the Notification has been received;
}
If(…) {…..…}
}

Algorithm Part 6: This part of algorithm is invoked by the WSC when unable to connect to a WSG.

{
*If(**Unable to Connect to WSG**) Then{*
*Send to Request Director a **WSG Recovery Request**;*
//WSG will execute algorithm part 4;
}
*If(Conneting to WSG **Timeout**) Then{*
If(Timeout over 3 times) Then{
*Set the status to **Unable to Connect to WSG**;*
}else{
Reconnecting to WSG;
}
}
}

The algorithm works in a-synchronized model so that after restart WSG can accept the WSCs' requests immediately without waiting for the recovery process to finish. Hence, the WSG's recovery is almost invisible to WSCs. As the recovery process for each WSC is handled separately, different WSCs will not affect each other during the recovery stage. Moreover, the algorithm works well with the WSG cluster model which will be discussed in next section. In Platform EGO, all WSGs have a share file system to save the configuration and the Recovery Tables so that if a WSG cannot restart somehow, its Recovery Table can be taken over by another WSG. Hence, WSGs can be backups for each other in cluster model.

5. CONCLUSION

The paper discussed the design and implementation of the web service gateway for Platform EGO. The pattern of Web Service Gateway to trap and map service requestors to its target services can be found in other commercial products, like WebSphere Application Server. Compared with these products, Platform EGO WSG is an enhanced implementation, as it provides more advanced functionalities, such as notification, fault-tolerance, and can work in cluster model with dynamic load balancing.

In the paper, the application level rollback recovery approach is used in WSG. Since the most critical information in a WSG is the active sessions between WSG and its WSCs, and the sessions between WSG and EGO internal services, we designed a session based a-synchronized recovery algorithm which has short freezing time and is able to isolate the recoveries of different WSCs. This fault tolerant approach can rebuild the service sessions and the notification mechanism after restart, to handle the WSG failures. Moreover, a lightweight notification mechanism is implemented to enable the EGO services to send messages to web service clients (WSCs) in a-synchronized model without any change to the underneath SOAP stack.

6. ACKNOWLEDGEMENT

The authors would like to acknowledge support from the China NSF under Grant No.60703024, Platform Computing Inc. under Grant 3B6056721421, and Jilin Department of Science and Technology under Grant No.20070122 and 20060532.

7. REFERENCE

[1] I. Foster (2006) Globus Toolkit Version 4: Software for Service-Oriented Systems. IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13

[2] Aghdaie, N., Tamir, Y. (2002) Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support. Proceedings of the IEEE International Conference on Computer Communications and Networks, pp 63-68

[3] Dialani, V., Miles, S., Moreau, et al (2002) Transparent Fault Tolerance for Web Services Based Architectures. Proceedings of 8th International Euro-Par Conference on Parallel Processing, Paderborn, Germany Proceedings. Volume 2400

[4] Giuliana Teixeira Santos, Lau Cheuk Lung, Carlos Montez (2005) FTWeb: A Fault Tolerant Infrastructure for Web Services. Proceedings of the 2005 Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)

[5] Alvisi, L. Bressoud, T.C. El-Khashab, and et al (2001) Wrapping server-side TCP to mask connection failures. Proceedings of INFOCOM 2001.Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies

[6] P. Townend and J. Xu (2004) "Replication-based Fault Tolerance in a Grid Environment", in Proceedings of U.K. e-Science 3rd All-Hands Meeting, Simon J. Cox Eds., Nottingham Conference Center, U.K., 31st Aug. - 3rd Sept., 2004, ISBN 1-904425-21-6.

[7] Condor Team, University of Wisconsin-Madison. (2002) Condor Version 6.8.2 Manual, http://www.cs.wisc.edu/condor/manual/v6.8/ref.html. Accessed Jan. 2006

[8] Anthony Nadalin IBM, Chris Kaler Microsoft, Ronald Monzillo Sun, et al (2006) wss-v1.1-os-UsernameTokenProfile. http://docs.oasis-open.org/wss/v1.1/. Accessed May 2006

[9] Steve Graham, Peter Niblett, Dave Chappell et al (2004) Publish-Subscribe Notification for Web services, 1.0. http://www-128.ibm.com/developerworks/library/ws-pubsub/WS-PubSub.pdf Accessed May 2006