Providing Reliable Web Services through Active Replication

Xinfeng Ye
Department of Computer Science, Auckland University, New Zealand
xinfeng@cs.auckland.ac.nz

Abstract

This paper presents a middleware that provides reliable web services using the active replication technique. The middleware uses a timestamp-based protocol to maintain the consistency of the replicas' states. Compared with the optimistic active replication protocol and group communication primitives, the protocol used in this paper reduces the replication overhead for a class of applications.

1. Introduction

Web services are self-contained, modular applications that can be located and invoked over the Internet. As more and more applications are built on web services, providing reliable web services is becoming an important issue. This paper presents a middleware that transparently supports reliable web services using the active replication technique. The middleware is responsible for maintaining the consistency of replicas' states. It uses a timestamp-based protocol to maintain the consistency of the replicas' states. Compared with the optimistic active replication protocol [4] and replication schemes based on the group communication primitives [12], the protocol used in this paper reduces the replication overhead for a class of applications.

The rest of the paper is organized as below. §2 describes the motivations for the project. §3 presents the middleware. §4 shows the performance of the middleware. §5 compares our system with other works. A conclusion is given in §6.

2. Motivations

In active replication, a system consists of several sites, called *replicas*. A client multicasts its request to all replica sites, which all handle the request and send back the response to the client. To ensure that the states on the replicas are consistent, it is required that (a) the code running on the replicas be deterministic, and, (b) the clients' requests must be sent to the replicas in the same order. Since multiple clients might send requests to replicas simultaneously, a *total order* is needed when multicasting clients' requests to the replicas. Total order means all requests are delivered to the replicas in the same order even if the senders of the requests are different. Group communication primitives [2] can be used to ensure the ordering of the clients' requests in active replication. However, replication schemes using group communication primitives suffer from high overhead due to the high synchronization cost amongst the replicas [4, 12].

In the optimistic active replication (*OAR*) [4], a sequencer periodically multicasts the ordering message (i.e. the sequence numbers) to all the replicas to inform them the order of executing the clients' requests. When a replica receives clients' requests, the replica does not execute the requests until it receives the ordering message from the sequencer. Since the replicas need to wait for the ordering messages from the sequencer before executing clients' requests, for some applications, it is possible to devise a scheme that achieves better performance than *OAR*. The system discussed in this paper uses a timestamp-based replication (*TSR*) scheme to guarantee that all the replicas execute the clients' requests in the same order. The experiment results show that *TSR* outperforms *OAR* in systems where the clients rarely send their requests to the systems simultaneously.

When a web service is replicated, to ensure the consistency of the replicas' states, some synchronization operations need to be carried out by replicas when the clients' requests are handled. Modifying the existing web services to incorporate the synchronization operations might introduce software bugs into the existing systems as well as make the existing systems less flexible. Thus, instead of modifying the existing web services, a middleware is used to manage the replication of the web services. The middleware uses the *TSR* protocol to handle the synchronization operations to ensure that the replicas' states are consistent.

6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007)
0-7695-2841-4/07 $25.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

3. The System

3.1. An Overview

The following assumptions are made about the system. The replicas communicate with each other through reliable FIFO channels. The replicas fail by crashing. The operations triggered by the clients' requests can be undone.

A web service is replicated at several sites. Each replica consists of a proxy web service site (PWSS) and a web service site (WSS). The WSS is a conventional web service provider. It hosts the code and data that provide the functionality of the web services. The PWSS is residing between the clients and the WSS. It is responsible for ensuring the consistency of the replica. Clients interact with the PWSSs. A client only needs to send its service request to one PWSS. The PWSSs are responsible for multicasting clients' requests to other replicas and returning results to the clients. To maintain the consistency of the WSSs' states, the PWSSs must ensure that all clients' requests are executed on the WSSs in the same order. The replicas form a group, called *service group*. Figure 1 shows the conceptual diagram of the replicas and the interactions between the components of the replicas.

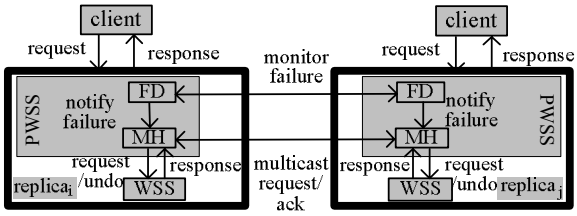


Figure 1 A Middleware for Replicated Web Services

When a client sends a service request, say *m*, *m* is sent to a PWSS. The PWSS multicasts *m* to all the other PWSSs for execution. The PWSSs use the *TSR* protocol to schedule the execution of *m*. The PWSSs forward *m* to their corresponding WSSs in the order determined by the *TSR* protocol. It is possible that it is found out later that *m* is executed by a WSS in a wrong order. In this case, the PWSS will instruct the WSS to undo the operation for *m* and re-execute *m* in the correct order. The WSSs return the responses to *m* to their corresponding PWSSs.

3.2 Design Details

3.2.1 The TSR Protocol The system is based on the state machine approach in [10]. The *TSR* protocol allows the replicas to reach an agreement on the order in which the clients' requests are processed. In *TSR*, each client's service request is given a unique timestamp. The replicas carry out the execution of the clients' requests in their timestamps' order. That is, a request with a smaller timestamp will be executed before a request with a larger timestamp. *TSR* assumes that it is rare that different clients send service requests to the replicas simultaneously. Thus, when a client's request is received by a PWSS, the PWSS sends the request to the PWSS's WSS for immediate execution. Meanwhile, the PWSS exchanges information with the other PWSSs to determine whether the request has been executed in the correct order. Requests executed in wrong order will be undone and reprocessed in the correct order. Since the execution of a client's request and the operation for determining the order in which the request should be carried out are being carried out concurrently, if the clients rarely send their requests to the replicas simultaneously, there is a high probability that their requests will be executed by the replicas in the correct order the first time. That is, very few undo operations will be carried out.

Each replica keeps a *logical clock* defined in [5]. The logical clock is an integer counter which increases monotonically. It is initialized to 0. A *timestamp* is a pair (*l_clock, ip*) where *ip* is the IP address of a replica and *l_clock* is the logical clock value of the replica. The ">" relation between two timestamps, (l_clock_1, ip_1) and (l_clock_2, ip_2), is defined below.

$(l\_clock_1, ip_1) > (l\_clock_2, ip_2) \Leftrightarrow (l\_clock_1 > l\_clock_2) \vee ((l\_clock_1 = l\_clock_2) \wedge (ip_1 > ip_2))$

The logical clock, *l_clock*, of a replica is updated according to **R1** and **R2** described below:

R1 **when** a replica, say *r*, receives a service request from a client:

$$l\_clock \leftarrow l\_clock + 1$$

R2 **when** a replica, say *r*, receives a multicast request, say *m*, from another replica:
let (*l_clock, ip*) be the timestamp of *m*, l_clock_r be the logical clock of *r*, and, ip_r be the IP address of *r*
if (*l_clock, ip*) > (l_clock_r, ip_r)
then $l\_clock_r \leftarrow l\_clock + 1$
else $l\_clock_r \leftarrow l\_clock_r + 1$

An *MsgList* on a replica holds the received clients' service requests before the requests are processed by the *WSS*. A *ProcessedList* is used to record the requests that have been executed by the *WSS*. *op*(*m*) denotes the operation that is invoked by the client's service request *m*. *l_clock* is the logical clock value of the PWSS and *ip* is the IP address of the PWSS. *ip* is also used as the ID of the PWSS.

R3	**when** receive a request, *m*, from a client:
1.	*m.init_receiver* ← *ip*

0-7695-2841-4/07 $25.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

2.	update *l_clock* according to **R1**
3.	*m.timestamp* ← (*l_clock, ip*)
4.	multicast *m* to all the replicas (including itself) in the service group

When a PWSS receives a client's request, say *m,* the PWSS sets the *init_receiver* attribute of *m* to indicate that the PWSS will be responsible for returning the response to *m* to the client (line 1 of **R3**). *m* is given a timestamp (line 3 of **R3**). Since *l_clock* increases monotonically (**R1** and **R2**), *m*'s timestamp is larger than the timestamps of any other requests on the PWSS. Then, *m* is multicast to all the other PWSSs. So that it can be executed on all the replicas.

R4	**when** receive a multicast request *m*:
1.	update *l_clock* according to **R2**
2.	generate an acknowledgment, *ack*, and, *ack.timestamp* ← (*l_clock, ip*)
3.	send *ack* to *m.init_receiver*
4.	**let** *wrong_set* = {*msg* | (*msg.timestamp* > *m.timestamp*) ∧ (*msg* is in *ProcessedList*)}
5.	**for each** *msg* such that *msg* ∈ *wrong_set* **do**
6.	(1) undo *op*(*msg*)
7.	(2) remove *msg* from *ProcessedList* and add *msg* to *MsgList*
8.	**end-for**
9.	add *m* to *MsgList* and sort *MsgList* into ascending order according to the timestamps of the messages in *MsgList*

When a PWSS, say *r*, receives a multicast request from another PWSS, *r* generates an acknowledgment message, *ack*, and assigns a timestamp to *ack* (line 2 of **R4**). According to line 1 of **R4** and **R2**, the timestamp assigned to *ack* is greater than the timestamps of all the requests previously received by *r*. *m*'s sender is *m.init_receiver* (line 4 of **R3**). *ack* is sent back to *m*'s sender (line 3 of **R4**). *ack* helps *m*'s sender to decide whether *m* has been executed in the correct order. Since all replicas should execute the requests in the order determined by the timestamps of the requests (i.e. the requests with smaller timestamps should be executed before the requests with larger timestamps), *r* needs to check whether any requests have been executed in a wrong order. Set *worng_set* contains all the requests that have been executed in a wrong order (i.e. the requests whose timestamps are greater than *m*'s timestamp and have been executed before *m* is received). For all the requests that have been executed in a wrong order, the operations triggered by these requests are undone (line 6 of **R4**) and these requests are added to *MsgList* for re-execution (line 7 of **R4**). After changes are made to *MsgList*, the requests in the list are re-sorted to ensure that they will be delivered to the WSS in ascending timestamp order (line 9 of **R4**).

R5	**when** receive an acknowledgment for message *m*:
1.	*m.ack* ← *m.ack* + *1*
2.	**if** (*m.ack* = *total*) **and** (*m* is in *ProcessedList*)
3.	send the result of *op*(*m*) to the client that sends *m*
4.	**fi**

m.ack (line 1 of **R5**) records the number of acknowledgments received for a multicast request *m*. *total* (line 2 of **R5**) represents the number of replicas in a service group. Assume that (a) a PWSS, say p_1, multicasts a request *m* to a PWSS, say p_2, and, (b) p_2 sends multicast messages $m'_1, \ldots, m'_n$ to p_1 before sending the acknowledgment for *m* to p_1. Since the communication channels between the replicas have the FIFO property, when p_1 receives *m*'s acknowledge from p_2, p_1 must have received $m'_1, \ldots, m'_n$ sent by p_2. According to line 4 - 8 of **R4**, when p_1 receives $m'_1, \ldots, m'_n$, p_1 has carried out the operations to ensure that *m* and $m'_1, \ldots, m'_n$ are executed in the correct order on p_1. In other words, if $m'_i.timestamp < m.timestamp$ where $1 \le i \le n$, p_1 would have scheduled m'_i to be processed before *m*. According to **R1** and **R2**, it can be seen that if p_2 multicasts a request *msg* after sending the acknowledgment for *m*, then *msg*'s timestamp must be greater than *m*'s acknowledgement's timestamp. Thus, if p_1 has received the acknowledgments for *m* from all the replicas, p_1 knows that it has scheduled to execute all the requests whose timestamps are less than *m.timestamp* before *m*. Hence, p_1 knows that *m*'s execution order is correct. This is because clients' requests are executed in their timestamps' order. As a result, if the WSS has completed the execution of *m*, p_1 can return the result of the execution to the client (line 2 – 4 of **R5**).

R6	**when** receive the result of *op*(*m*) from the *WSS*:
1.	cache the result of *op*(*m*), and, add *m* to the end of *ProcessedList*
2.	**if** ((*m.ack* = *total*) **and** (*m.init_receiver* = *ip*)) send the result of *op*(*m*) to the client **end-if**
3.	**if** *MsgList* is not empty
4.	**let** *fm* be the first request in *MsgList*
5.	remove *fm* from *MsgList* and send *fm* to the *WSS*
6.	**end-if**

When the WSS completes the processing of a client's request, the PWSS stores the result of the processing to cope with possible failure of the PWSS that receives the client's request. If the PWSS is

IEEE COMPUTER SOCIETY

responsible for sending the result back to the client (i.e. *m.init_receiver* = *ip*) and the PWSS has received the acknowledgments for the request from all the replicas, as explained for **R5**, the result of the processing can be sent to the client (line 2 of **R6**). Then, the next message in *MsgList* is sent to the WSS for processing (line 3 – 6 of **R6**).

After being processed by the WSS, the request messages are added to the *ProcessedList*. Thus, the list might grow infinitely. To avoid this problem, the PWSSs periodically broadcast a list of messages that have been acknowledged by all the PWSSs. For a message *m*, if *m* and all its predecessors in *ProcessedList* have been acknowledged, *m* and its predecessors can be removed from *ProcessedList*.

3.2.2 The Implementation of the System Requests sent by clients are encapsulated in SOAP messages. Each SOAP message has a unique ID. The ID is used for detecting possible duplicated clients' requests in the event of replica failure. When writing client applications, programmers use the classes in package *RWS* that is provided by the middleware to make calls to web services. These classes automatically generate the SOAP messages encapsulating calls to web services and assign IDs to the messages.

Each PWSS consists of two modules, i.e. a message handler (MH) and a failure detector (FD) as shown in Figure 1. A client application program interacts with an MH to exchange requests and responses. The MH is responsible for (a) handling the SOAP messages received from clients, (b) recording the IDs of the messages received from clients, (c) holding copies of the responses to clients' requests and sending responses back to the clients (if necessary), (d) running the *TSR* protocol to interact with the MHs of other PWSSs, and, (e) handling failure of the replicas. The FD is responsible for monitoring the failure of the other replicas. The FD is implemented as a class $\Diamond S$ failure detector [3].

The MH puts the received clients' requests in the *MsgList*. The MH sends the request in the *MsgList* to its corresponding WSS one at a time. That is, a request is not sent to the WSS until the response to the previous request in the list is received from the WSS. This ensures that the requests are executed by the WSS in the order determined by the *TSR* protocol.

It should be noted that a WSS might host many web operations. Only the service requests sent to the same operation need to be ordered. Requests sent to different operations do not need to be ordered. Thus, for each web operation offered by the WSSs, the MHs run a *TSR* protocol thread to multicast and order the requests sent to the operation. Each operation has its own *MsgList* set up on each of the MHs. Thus, requests for different operations can be sent to the WSSs simultaneously as long as these requests are ordered in their respective *MsgLists*.

When an MH receives a response from its WSS, it stores the response in its buffer in order to handle possible failure. An MH receives a client's request either (a) directly from the client or (b) from another MH. Case (b) occurs if the client sends the request to a different MH in the service group; and, consequently, the request is multicast to all other MHs in the service group. As a result, each MH also receives requests that are not directly sent to it by clients. When a response to a client's request is received, the MH that receives the client's request directly is responsible for returning the response to the client. After delivering the response to the client, the MH asks the other PWSSs to delete the response from their buffers.

Clients need to handle the failure of a replica in the sense that the clients need to connect to another replica in the service group. Package *RWS* includes classes that automatically handle replica failure for clients. Programmers only need to provide a configuration file specifying the addresses of the replicas in the service group when they use the classes in *RWS*. The information in the configuration file allows the methods of the classes in *RWS* find alternative replicas when a replica fails. The classes in package *RWS* provide all the mechanisms discussed below.

If a replica fails before a client sends its request, the failure is discovered when the client attempts to connect to the replica's PWSS and fails in its attempt. In this case, the client will send its request to another replica's PWSS.

In a synchronous web service call, a client connects to a replica's PWSS when it sends a request to the web service. If the PWSS fails, the client loses the connection to the PWSS. In this case, the client attempts to establish a connection with another replica's PWSS in the service group. After a client connects to another PWSS, the client resends its request. Re-sending the request is necessary. This is because the failed PWSS might fail before it multicasts the client's request to other PWSSs. However, the resending of the request might result in duplicated request since the failed PWSS might have multicast the client's request to other PWSSs before it fails. To cope with message duplication problem, when an MH receives a request, it uses the ID of the request to check whether the same request has been received previously. If the request has been received previously and been processed in the failure recovery phase (see later), the MH does not multicast the request to the other replicas. In this case, the response to the request will be sent to the client when the response is available.

IEEE COMPUTER SOCIETY

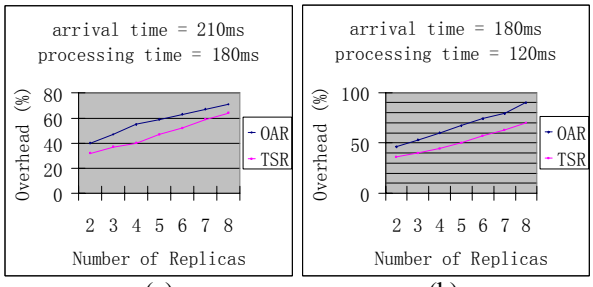
The FDs monitor whether a replica fails by exchanging messages with each other. When a replica fails (i.e. the replica's FD does not respond to other FDs' messages), the PWSSs enter the recovery phase. During the recovery phase, the replicas do not accept any client request. In the recovery phase, the MHs run the consensus protocol in [3, 4] to agree on the execution order of the clients' requests that have been received by the MHs. These requests are marked as having been sorted. When the responses to these requests become available, the replicas that receive the requests from the clients directly can send the responses to the clients immediately. The recovery phase ends after the MHs reach agreement on the execution order of the clients' requests.

4. Performance

Experiments were carried out on machines to measure the overheads of the PWSS when a service group is deployed over a WAN. In the experiments, the replicas are connect by Ethernet. To simulate that the replicas are connected by a WAN, each message exchanged amongst the PWSSs is held in a buffer for 120ms before it is processed (i.e. the WAN has a one-way delay of 120ms). Overhead is defined as $(t_n - t_1)/t_1$ where t_n is the service response time when the service group has n replicas and t_1 is the service response time when no replication is used.

In the following discussion, *arrival rate* means the inverse of the interval between the clients' requests arrive at the system while *processing rate* is the inverse of the duration to process a client's request by the WSS. The arrival rate and the processing rate are both simulated to follow an exponential distribution. The generated clients' requests are randomly distributed to the replicas. It is assumed that the arrival rate is less than the processing rate. That is, the system has sufficient processing capacity to handle the clients' requests.

The first experiment compares the performance of the *OAR* and the *TSR* when they are used by the PWSS to ensure the consistency of the replicas. In the *OAR*-based implementation, the clients are modified to send their requests to all the PWSSs instead of a single PWSS. Also, the clients will collect replies from all the PWSSs for each of their requests. The experiments assume that it is rare that different clients send requests to the PWSSs simultaneously. Figure 2(a) and Figure 2(b) show the performance of the system under different mean arrival interval and processing time. It can be seen that the *TSR* protocol outperforms the *OAR* protocol in both cases. It also can be seen that, when the mean processing time increases, the overhead decreases due to the reduced synchronisation cost (i.e. the cost for ordering the clients' requests amongst the replicas) for per unit of computation. Since the clients hardly send requests to the PWSSs simultaneously, the number of requests that are executed in wrong order when using the *TSR* protocol does not increase significantly as the number of replicas increases. As a result, the costs for undoing and re-executing the requests that were executed in the wrong order do not increase significantly. Thus, the synchronisation costs in running the *TSR* protocol dominate the overheads. For both protocols, as more replicas are added to the systems, it takes longer to run the protocols amongst all the replicas in the system. As *OAR* and *TSR* both require one round of multicasting and one round of acknowledging/sending response, the costs of running the two protocols increase at roughly the same rate. Hence, it can be seen that the differences in the overheads of the protocols are about 10% and 20% in Figure 2(a) and 2(b) respectively.



(a) (b)

Figure 2 Comparing the Performance of *TSR* and *OAR*

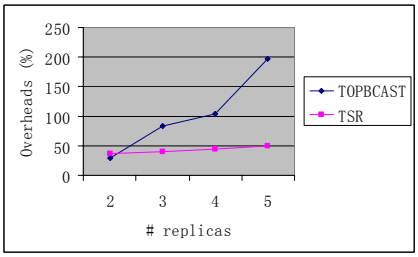


Figure 3 Comparison of *TSR* and *TOPBCAST*

The middleware in [12] uses a group communication protocol, i.e. the *TOPBCAST* protocol, to ensure the consistency of the replicas' states. Experiments were carried out to compare the performance of the middleware in [12] with the middleware discussed in this paper. Figure 3 shows the performance of the two middleware. From the figure, it can be seen that the two approaches have similar overheads when there are two replicas in the system. However, when there are more than two replicas in the

IEEE COMPUTER SOCIETY

system, the overhead of the middleware in this paper is significantly lower than the one in [12].

5. Related Work

Ye et al. [12] proposed the use of a middleware for managing the replicated web services. Salas et al. [9] described a similar scheme for web services replication. [9] and [12] both use a group communication protocol to ensure the consistency of the replicas' states. Generally speaking, group communication protocols incur high synchronization overheads [4, 12]. Our experiment shows that the middleware based on the *TSR* protocol outperforms the one based on the *TOPBCAST* protocol.

Clustering has been used to achieve scalability and availability in enterprise applications [1, 6]. Machines in cluster architecture are normally located close to each other and are connected by either proprietary hardware link/bus-interconnection or LAN-based interconnects. A load balancer distributes the clients' requests to the machines in the cluster. It is pointed out in [7] that (i) network latency is one of the factors that affect an application's response time, and, (ii) one way to deal with the impact of latency is to locate web application geographically closer to users. Therefore, instead of hosting an application at a single site, the application might be replicated at several locations that are geographically far apart from each other. As a server cluster is normally deployed at a single site, the system proposed in this paper can be used to complement the cluster architecture for web applications. The system in this paper can be used to ensure that the requests for executing operations are delivered to the replicas at different geographical locations in the same order. At each location, a server cluster can be used to execute the received requests. As discussed earlier, for each operation, in order to guarantee the execution order, the requests for carrying out the operation have to be executed one by one. However, the requests for different operations can be carried out concurrently by the server cluster. Thus, deploying a server cluster at each replication site can still improve the overall performance of the site.

Sousa et al. [11] and Patino-Martinez et al. [8] proposed schemes for executing the multicast messages optimistically. Like the *OAR* protocol, their schemes use a sequencer to (a) determine the order in which the multicast messages should be executed, and, (b) multicast the sequence numbers to all the replicas. However, they allow the multicast messages to be tentatively processed before receiving the sequence numbers from the sequencer. Compared to the schemes in [11] and [8], the *TSR* protocol does not use a sequencer to order the multicast messages. Thus, the *TSR* protocol does not concentrate the communication generated by multicasting the sequence numbers at any replica. Instead, the communication load is shared amongst all the replicas.

6. Conclusion

The system discussed in this paper supports reliable web services based on active replication. As the system is designed as a middleware, it allows existing web services to be replicated at different sites without modifying the existing code. This allows web services to be replicated easily. The system uses a timestamp-based active replication protocol. Empirical data show that the protocol outperforms the *OAR* protocol and group communication primitives when clients rarely send their requests to the system simultaneously.

References

[1] BEA (2005). Using WebLogic Server Clusters, Available from: http://e-docs.bea.com/wls/docs70/cluster/index.html, Accessed 26th January 2007

[2] K. P. Birman, Building Secure and Reliable Network Applications, Manning Publications Co., 1996

[3] Chandra, T.D. and S. Toueg, Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, 1996. **43**(2): p. 225-267

[4] P. Felber, A. Schiper, Optimistic active replication, Proc. Of 21st International Conference on Distributed Computing Systems, pp333 – 341, 2001

[5] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, CACM, Vol. 21, No. 7, pp558-565, 1978

[6] Li, S. (2003). High-impact Web tier clustering, Available from: http://www-128.ibm.com/developerworks/java/library/j-cluster1/, Accessed 26th January 2007

[7] Microsoft ACE Team (2003), Performance Testing Microsoft .NET Web Applications, Microsoft Press

[8] A. Patino-Martinez, R. Jimenez-Peris, B. Kemme, G. Alonso: MIDDLE-R: Consistent database replication at the middleware level. ACM Trans. Comput. Syst. 23(4): 375-423, 2005

[9] J. Salas, F. P´erez-Sorrosal, M. Patino-Martnez, R. Jimenez-Peris, WS-Replication: A Framework for Highly Available Web Services, Proc. of the 15th Intl. Conf. on World Wide Web, pp.357-366, 2006

[10] Fred B. Schneider: Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. ACM Comput. Surv. 22(4): 299-319, 1990

[11] A. Sousa, J. Pereira, F. Moura, R. Oliveira, Optimistic Total Order in Wide Area Networks, Proc. 21st IEEE Symp. on Reliable Distributed Systems, 190-199, 2002,

[12] Ye, X., & Shen, Y., A middleware for replicated Web services, Proceedings of 2005 IEEE International Conference on Web Services, pp. 631-638, 2005

0-7695-2841-4/07 $25.00 © 2007 IEEE

IEEE COMPUTER SOCIETY