A Lightweight Fault Tolerance Framework for Web Services*

Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University, 2121 Euclid Ave, Cleveland, OH 44115
wenbing@ieee.org

Abstract

In this paper, we present the design and implementation of a lightweight fault tolerance framework for Web services. With our framework, a Web service can be rendered fault tolerant by replicating it across several nodes. A consensus-based algorithm is used to ensure total ordering of the requests to the replicated Web service, and to ensure consistent membership view among the replicas. The framework is built by extending an open-source implementation of the WS-ReliableMessaging specification, and all reliable message exchanges in our framework conform to the specification. As such, our framework does not depend on any proprietary messaging and transport protocols, which is consistent with the Web services design principles. Our performance evaluation shows that our implementation is nearly optimal and the framework incurs only moderate runtime overhead.

1. Introduction

Many Web intelligence systems offer their services in the form of Web services, and some of the core services must be made highly available and reliable to accomplish the systems' missions. In fact, the capability of automatically reconfiguring themselves for continuous operation in the occurrences of component failures should be an essential element of any intelligence system. However, designing a sound fault tolerance solution for Web services is not trivial.

It is attempting to perform a relatively straightforward translation of many existing fault tolerance mechanisms for the older generations of distributed computing platforms such as those described in FT-CORBA [18] to Web services. We argue against such an approach, for several reasons. As pointed out by many experts and researchers in distributed computing, Web services is drastically different from older distributed computing technologies [20, 24] in that Web services is designed for Web-based computing over the Internet, and it adopts a message based approach for maximum interoperability, while the older technologies are not designed for Internet and primarily focus on Application Programming Interface (API) based interactions. Furthermore, Web services advocates flexibility, composability, and technology independence. Hence, a fault tolerance solution for Web services must take an approach that is consistent with the design principles of Web services. Secondly, the FT-CORBA standard [18], which is one of the major outcomes of the fault tolerance research for CORBA, contains a great number of APIs for replication and fault management, and many sophisticated mechanisms, which has been considered too heavy weight even for CORBA applications, let alone for Web services.

The above observation prompted us to design a novel, lightweight fault tolerance framework for Web services. The framework has the following features:

- It does not use any proprietary communication protocol to coordinate the server replicas, and to enable the communication between the clients and server replicas. All the messaging required for replication is defined in WSDL which is publicly visible to users.
- This decision leads us to adopt a consensus-based algorithm [16], rather than a group communication system, to perform state machine based replication. The algorithm ensures the total ordering of all requests to the replicated Web service, and a consistent membership view of the replicas (which is crucial to avoid the split-brain syndrome [6]).
- The framework is backward compatible with the WS-ReliableMessaging [4] specification, which ensures reliable point-to-point communication of Web services. A Web service using our framework can be protected against failures by replication when needed, otherwise, it runs as a WS-ReliableMessaging implementation. The switch between replication and non-replication modes can happen dynamically during runtime. Unlike other fault tolerance framework, our framework does not incur any extra overhead when running in non-replication mode (*i.e.,* single replica).
- Our framework is lightweight in that it does not impose sophisticated replication and fault management

*This work was supported by Department of Energy Contract DE-FC26-06NT42853, and by Cleveland State University through a Faculty Research Development award.

requirement, as did in FT-CORBA. The configuration is through a simple property file. The fault detection is incorporated in the replication mechanisms.

We have implemented our framework using Apache Axis2 [2] (the latest open source SOAP engine) and Sandesha2 [3] (an open source implementation of the WS-ReliableMessaging specification on top of Axis2). The consensus-based replication algorithm is adapted from the BFT algorithm [7]. It is essentially an implementation of the Paxos algorithm [16]. The performance of the framework is carefully characterized and optimized. The runtime overhead is quite moderate considering our all-Web-services-technology approach.

2. Related Work

A considerable number of high availability solutions for Web services have been proposed in recent years. Two of them, namely, WS-Replication [22] and Thema [19], are most closely related to this work because they both ensure strong replica consistency for Web services.

Similar to our work, WS-Replication achieves consistent replication of Web services by totally ordering all incoming requests to the replicated Web service. Even though the interfaces to the client application and the replicated Web services conform to the Web services standards, the actual transport is carried out using JGroup [14], which is a proprietary group communication system. JGroup does offer a SOAP transport. However, the performance is poor when such a transport is used. Consequently, proprietary requests serialization is used to achieve decent performance. Unfortunately, such a move is orthogonal to the Web services design principles, which insists on standard Internet based transport protocols. The use of a proprietary group communication system is also problematic, because the clients and all replicas are strongly coupled to a single technology, which would pose interoperability problems. From the implementation perspective, WS-Replication uses separate proxy and dispatcher processes to capture and multicast clients' requests, and to receive multicast messages from JGroup and forward the requests to the replicated Web services, respectively, which is inefficient. Our framework avoids the above problems by using standard Web services transport and messaging protocols for all interactions between clients and the Web services, and among the replicas. Furthermore, in our framework, clients communicate directly with the replicated Web services.

Thema [19] reported a Byzantine fault tolerant [15] framework for Web services. Even though it is also constructed on a consensus-based replication algorithm like ours, an adaptor is used to interface with an existing implementation of the algorithm [7] which is based on UDP multicast, rather than the standard SOAP/HTTP transport, as such, it suffers from the same problem of WS-Replication [22]. It does, however, use a much weaker fault model [15].

Other work [5, 8, 9, 10, 11, 12, 17, 21] either uses a different approach such as checkpointing and replay, or is still in conceptual stage. Some of the work ignored the consistency issues when performing replication and failure detection over the Internet, which may be problematic because the Internet is largely an asynchronous system.

3. System Models

We consider a Web service and its clients interacting over the Internet. When considering the safety of our replication algorithm, we use the asynchronous distributed system model. However, to ensure liveness, certain synchrony must be assumed. Similar to [7], we assume that the message transmission and processing delay has an asymptotic upper bound. This bound is dynamically explored in our algorithm in that each time a view change occurs, the timeout for the new view is doubled.

We assume a crash fault model, *i.e.,* a Web service replica might fail due to hardware or software failures, but once it fails, it stops emitting any messages. In particular, neither the clients nor the replicas behave maliciously. We assume that the network may incur transient faults, but they can be promptly repaired, *i.e.,* we assume network partition does not occur.

The Web service is replicated using a state-machine based approach, and hence, we assume the Web service operates deterministically. We are aware that most practical Web services contain some degree of nondeterminism. How to fully cope with such nondeterminism systematically is beyond the scope of this paper.

We assume that $2f + 1$ replicas are available, among which at most f can be faulty. Similar to [7], each replica is assigned a unique id i, where i varies from 0 to $2f$. For view v, the replica whose id i satisfies $i = v \ mod \ (3f + 1)$ would serve as the primary. The view starts from 0. For each view change, the view number is incremented by one and a new primary is selected. Due to space limitation, we omit the discussion on how to cope with the scenarios of adding more replicas to the system, or when the number of replicas is reduced below the threshold for the remaining replicas to form a quorum.

4. Replication Algorithm

The replication algorithm in our framework is adapted from the BFT algorithm by Castro and Liskov [7]. Due to the stronger fault model used in this work, the algorithm is significantly simplified, especially the view change algorithm.

4.1. Normal Operation

The normal operation of the algorithm is shown in Figure 1. When the client issues a request to a replicated Web service, the request is multicast to all replicas. The request has the form $<\text{REQUEST}, s, m, o>$, where s is a unique sequence id, m is the message number within the sequence s, and o is the operation to be invoked on the Web service,

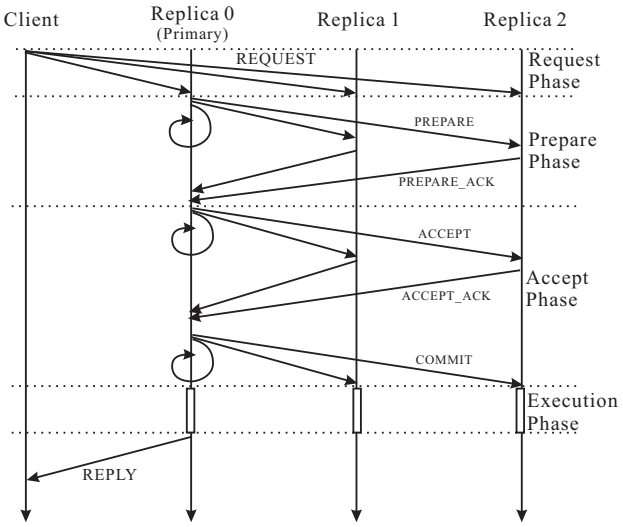

Figure 1. Normal operation of the replication algorithm.

together with necessary parameters. On receiving a client's request, a replica checks if it is a duplicate. The primary retrieves the corresponding response from its log (if one can be found) and sends it to the client if the request is a duplicate. The duplicate request is dropped subsequently. The backups simply drop the duplicate without resending the response for efficiency reason.

Note that the message format described here captures the essential information needed for total ordering. The actual message is an XML document encoded according to the SOAP standard. The concept of sequence is introduced in WS-ReliableMessaging [4]. When the client sends its first request to a Web service via WS-ReliableMessaging, a unique sequence is established between the client and the Web service. Every reliable message sent over the sequence is assigned a message number, which starts from 1 and increases by 1 for each subsequent message sent. A sequence forms a unidirectional reliable channel between two communicating parties. Therefore, another sequence is established for the Web service to send the reply back to the client. The mechanisms for establishing and terminating a sequence is elaborated in the WS-ReliableMessaging specification [4], and hence, they are not repeated here.

When a replica accepts a client's request (to distinguish from the control messages used to establish total ordering, the client's requests are referred to as application requests from now on), and it is the next expected message from its sequence, the replica starts a view change timer. The timeout is set to allow the consensus to be reached on the ordering of the message.

When the primary p (replica 0 in the figure) is ready to order this message, it assigns the message a monotonically increasing sequence number n (not to be confused with the sequence concept in WS-ReliableMessaging) and its current view number v, and multicasts a prepare request to all replicas (the one to its own is not actually sent to the network - it is stored in the local data structure). This starts the prepare phase.

The prepare message has the form $<\text{PREPARE}, v, n, s, m>$, where v is the current view number, n is the global sequence number assigned by the primary for the application request message identified by s and m. A backup accepts a prepare message provided the replica is in view v and it has not accepted a prepare request for the same application message in view v.

Note that a backup might receive a prepare request ahead of the application request being ordered. As long as the sequence between the backup and the client is open, the backup will eventually receive the request. If the sequence has been terminated due to a premature timeout at the backup, the backup reestablish the sequence and asks the primary for retransmission of the message. The missing of the message being ordered does not prevent a backup from accepting the prepare request.

When a backup accepts the prepare request, it stores the message in its data structure and sends a prepare response to the primary. The prepare response has the form $<\text{PREPARE\_ACK}, v, n>$. At this point, we say the replica has *prepared* the application request with sequence number n.

When the primary receives a prepare response, it verifies that the response indeed contains a matching sequence number and the view number with the prepare request it has sent. It logs a valid prepare response to its data structure. When the prepare response messages from different replicas and its own prepare request form a quorum, *i.e.,* the total number of such messages is equal to $f + 1$, the primary multicasts an accept request, in the form $<\text{ACCEPT}, v, n>$, to all replicas. This enters the accept phase.

When a backup receives an accept request, it accepts the request provided that it is in view v and it has accepted the corresponding prepare request in view v. The backup logs the accept request and sends an accept response $<\text{ACCEPT\_ACK}, v, n>$ to the primary. The backup is said to have *accepted* the application message with n in view v at this point.

When the primary receives an accept response message, it verifies the message by matching the view and sequence number with those in the accept request. It logs a valid accept response until a quorum is formed (including the accept request it has sent). The application request with sequence number n is now both *accepted* and *committed* at the primary. The application request can be delivered if all previous requests have been delivered to the Web service.

Before a backup could deliver and execute the application request, however, it must be sure that a quorum of replicas have agreed on the ordering for the message. This requires the primary to disseminate a commit message $<\text{COMMIT}, v, n>$ to all backups when it has collected f accept responses from different backups. This commit message is acknowledged in the transport level (by the WS-

ReliableMessaging mechanism) instead of the algorithm level.

On receiving the commit message, a backup knows that the application request with sequence number n is *committed*, and it is ready to deliver the application request being ordered if it has delivered all previous requests to the Web service.

When the primary finishes executing the application request, it logs the corresponding reply and sends the message to the client. For performance reason, a backup only logs the reply and does not actually send it to the network, unless the replica becomes a new primary after a view change. The logged responses will be garbage collected when the client acknowledges them.

4.2. Garbage Collection and Checkpoint

A replica must keep the application requests in its log until all non-faulty replicas have delivered them. To avoid holding on the requests forever, each replica periodically takes a checkpoint of its state according to a deterministic algorithm (say, take one checkpoint for every 10 requests executed).

After taking a checkpoint, a replica multicasts a checkpoint message to all other replicas. The checkpoint message has the form $<$CHECKPOINT$, n, i>$, where n is the sequence number of the last application request executed, and i is the replica id. If a replica has collected a quorum (*i.e.,* $f + 1$) of checkpoint messages from different replicas (including the message it has sent) for n, the checkpoint for n is said to have become *stable*, and the replica garbage collects all logged messages up to n, and the associated control messages (prepare, accept and commit etc.). It also deletes all previous checkpoints. A backup might lag behind and needs an application request that has been garbage collected by the primary, in which case, it asks the primary for a state transfer instead.

4.3. View Change

If a backup i could not advance to the *committed* state on expiration of the view change timer, it initiates a view change by sending a view change message to the new primary in view $v + 1$, as shown in Figure 2. The view change message has the form $<$VIEW_CHANGE$, v + 1, l, P, i>$, where l is the sequence number for the last stable checkpoint known to i, P is a set of accepted records. Each accepted record consists of a tuple $<view, n, s, m>$, where $view$ is v or smaller.

When the primary in view $v + 1$ has collected $f + 1$ view change messages, including the one it would have sent, it installs a new view and notifies the backups with a new view message $<$NEW_VIEW$, v + 1, O>$, where O is a set of prepare messages. The prepare messages included in O is determined in the following way:

– If the new primary received an accepted record $<view, n, s, m>$ in a view change message (including the one it would have sent), it constructs a new prepare message $<$PREPARE$, v, n, s, m>$.

– There might be a gap between the sequence number of the last known checkpoint, and the least sequence number of an accepted record, or a gap between two accepted records, in which case, a prepare message is created for a null application request, *i.e.,* $<$PREPARE$, v, n, s^{null}, m^{null}>$. The execution of the null application request is a no-op (*i.e.,* there is no actual execution for the null message).

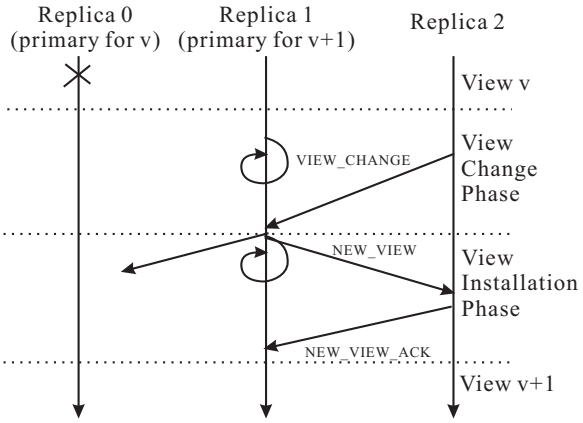

Figure 2. Sequence diagram showing the steps of the view change algorithm.

When a backup receives the new view message, it accepts the message if it has not installed a newer view. If the replica accepts the new view message, it installs the new view, and processes the prepare requests included in the new view message as usual.

4.4. Informal Proof of Correctness

We now provide an informal proof of the safety of our replication algorithm. Due to space limitation, the proof for liveness is omitted.

Lemma 1: For all replicas that commit an application request r in the same view v, they agree on the same sequence number n.

We prove by contradiction. Assume that two replicas i and j committed the same application request r with two different sequence numbers m and n, respectively. For replica i to commit the request with m, it must have received a commit request from the primary (or it has sent a commit request if it is the primary itself). This means that a quorum of $R1$ replicas have accepted assignment of the sequence number m for r. Similarly, because j committed r with a different sequence number n, a quorum of $R2$ replicas must have accepted the assignment of the sequence number n for r. By definition of quorum, $R1$ and $R2$ must intersect in at least one non-faulty replica, which implies this replica accepted two different sequence numbers for the same request r. This contradict to our algorithm because a

non-faulty replica accepts only one sequence number for each request. Therefore, lemma 1 stands.

Lemma 2: For replicas that commit an application request r in different views, they agree on the same sequence number.

We prove by contradiction. Assume that replica i committed r with a sequence number m in view v, and replica j committed r with a different sequence number n in view u. Without loss of generality, we assume $u > v$. Since i committed r with m in view v, there are a quorum of $R3$ replicas that have accepted the sequence number assignment for r. To install a new view u, the new primary must have collected the view change messages from a quorum of $R4$ replicas. $R3$ and $R4$ must intersect in at least one non-faulty replica. Since this replica has accepted the sequence number assignment for r, it would have included the accepted tuple in its view change message sent to the new primary, and the new primary must have constructed a prepare message using the sequence number m for r. If j committed r in view u, it must have accepted the accept message for the m and r binding, which contracts our assumption. Therefore, lemma 2 is correct.

Theorem: The safety property holds for the replication algorithm.

Due to lemma 1 and lemma 2, for all replicas that commit a request r in any view, they agree on the same sequence number for r. Therefore, the safety property holds for the replication algorithm.

5. Implementation and Performance

We have implemented our fault tolerance framework by extending Sandesah2. The architecture of the server-side framework is shown in Figure 3. The InOrder Invoker component in Sandesha2 is replaced by a Total Order Invoker, which is responsible to deliver requests to the replicated Web service in a total order. At each replica, the Total Order Invoker polls the replication engine for the next application request to be delivered, then it fetches the application message from the In Msg Queue, which stores all incoming application requests.

The Sender component in Sandesha2 is replaced by a multicast-capable Sender. On the server side, the multicast is used only by the primary, and by a backup for the checkpoint messages. To perform the multicast, multiple threads are launched to concurrently send the same message to different destinations using Axis2. Each thread is responsible to send the message to a distinct destination point-to-point.

The In handler in Sandesha2 is augmented to handle the control messages used by the replication algorithm (*i.e.,* prepare, prepare response, accept, accept response, commit, checkpoint, view change, and new view messages). All incoming messages are placed in the In Msg Queue after preliminary processing, and before they are delivered to the Web service by the Total Order Invoker.

The Out handler in Sandesha2 is largely left intact. All outgoing messages are placed in the Out Msg Queue before they are sent out by the Sender component.

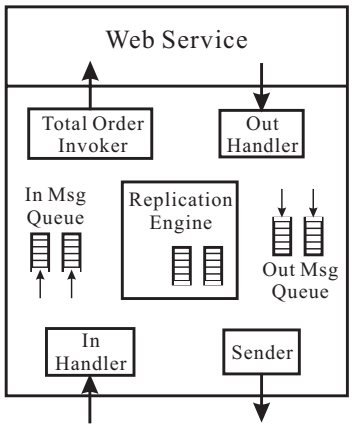

Figure 3. The architecture of the server-side framework.

The sending of the replication control messages is carried out using the normal Axis2 interface, which means such messages will be treated as application messages by the Sandesha2 mechanisms, except that some messages are multicast by the Sender component. The Sender knows what messages to be multicast by examining the SOAP action property included in each message. If a multicast is needed, the Replication Engine is consulted to obtain the multicast destinations.

The Replication Engine is the core addition to the Sandesha2 framework. This component drives the execution of the replication algorithm. The Replication Engine uses its own storage to log the replication control messages, and the checkpoints.

The client-side architecture is similar, except that the application replies are not totally ordered (they are FIFO ordered within each sequence according to WS-ReliableMessaging, however). The Replication Engine simply keeps the server-side configuration information so that the Sender knows where to multicast the application requests.

Even though in the replication algorithm description, each application request is assigned a unique sequence number, doing so would be very inefficient. Similar to the BFT framework, we incorporated a batching mechanism to improve performance. The batching mechanism works in the following way. At the primary, it does not immediately order an application request when it is in FIFO order within its sequence, instead, it postpones doing so if there are already k batches of messages being ordered, where k is a tunable parameter and it is often set to 1. When the primary is ready to order a new batch of messages, it assigns the next sequence number for a group of application requests, at most one per sequence, and the requests ordered must have been FIFO ordered within their own sequences.

Our performance evaluation is carried out on a testbed consisting of 12 Dell SC440 servers connected by a

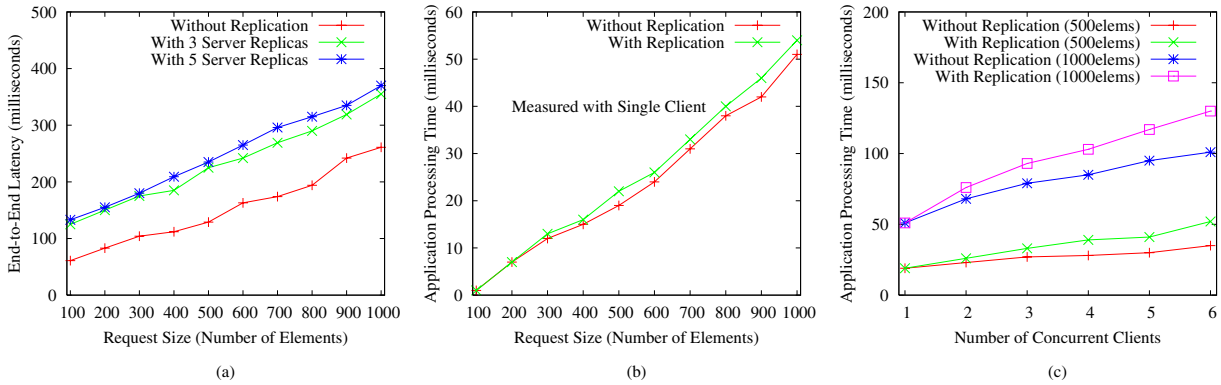

Figure 4. Latency measurement results. (a) End-to-end latency of the echo operation for replication degrees of 1, 3 and 5. (b) Application processing latency for different request sizes. (c) The application processing time when different number of concurrent clients are present.

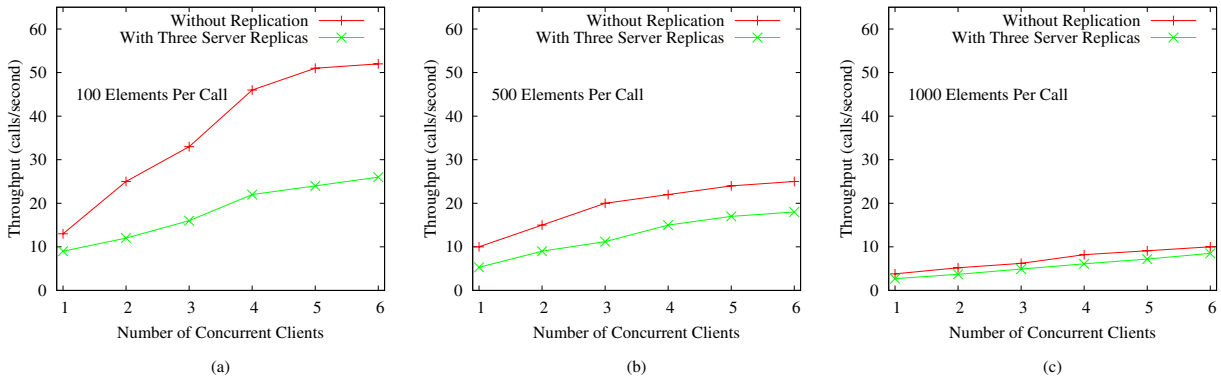

Figure 5. System throughput with and without replication. (a) Each request contains 100 elements. (b) Each request contains 500 elements. (c) Each request contains 1000 elements.

100Mbps Ethernet. Each server is equipped with a single Pentium D 2.8GHz processors and 1GB memory running SuSE 10.2 Linux.

We focus on reporting the runtime overhead of our replication algorithm during normal operation. A backup failure virtually does not affect the operation of the algorithm, and hence, we see no noticeable degradation of runtime performance. However, when the primary fails, the client would see a significant delay if it has a request pending to be ordered, due to the timeout value for view changes. The timeout is usually set to 2 seconds in our experiment which is in a LAN environment. In the Internet environment, the timeout would be set to a higher number. If there are consecutive primary failure, the delay would be even longer.

A simple echo test application is used to characterize the runtime overhead. The client sends a request to the replicated Web service and waits for the corresponding reply within a loop without any "think" time between two consecutive calls. The request (and the reply) contains an XML document with varying number of elements, encoded using AXIOM (AXis Object Model) [1]. At the replicated Web service, the request is parsed and a nearly identical reply XML document is returned to the client.

In each run, 1000 samples are obtained. The end-to-end latency for the echo operation is measured at the client. The latency for the application processing time (to parse the request and to generate a reply) and the throughput are measured at the replicated Web service. In our experiment, we vary the number of replicas, the request sizes in terms of the number of elements in each request, and the number of concurrent clients.

Figure 4 shows the latency measurement results. In Figure 4(a), The end-to-end latency of the echo operation is reported for replication degrees of 1, 3 and 5. When there is only a single replica, our framework rolls back to the Sandesha2 implementation without incurring any additional overhead, which is why the latency is significantly smaller than other scenarios.

The latency incurred by our replication algorithm for three-way replication ranges from 60ms for short requests to about 100ms for large requests that requires substantial processing, as shown in Figure 4(b). The increase in latency for longer requests (or more accurately, more complex requests) is likely due to the CPU contention for process-

ing of the application requests (by the Web service) and the replication mechanisms (by our framework), as suggested by Figure 4(b), which shows a small increase in application processing time for each request when replication is enabled comparing with the non-replication case. The CPU contention is more obvious when there are concurrent clients, as shown in Figure 4(c). For both Figure 4(b) and (c), varying replication degree from 3 to 5 does not cause noticeable change in latency.

The throughput measurement results for different request sizes are shown in Figure 5. It can be seen from Figure 5(a), for short request sizes, the throughput degradation when replication is enabled is significant, especially when there are many concurrent clients. This is not surprising considering the complexity of the replication algorithm. Even with optimal batching for 6 concurrent clients, the primary must send 3 control messages and receive 6 control messages (2 of them in the transport level) to order the 6 application requests. The approximately 50% reduction in throughput is nearly optimal. [22] reported a 2/3 reduction in throughput when the standard SOAP protocol and Web services are used, which is less efficient due to their architecture. When the application request complexity is increased, the throughput reduction becomes less, as shown in Figure 5(b) and (c).

6. Conclusions

In this paper, we presented the design and implementation of a fault tolerance framework for Web services. It uses 100% Web services technology to build the fault tolerance service, and hence, it is more preferable to achieve interoperability. It does not require sophisticated replication and fault management, and it does not rely on bulky group communication systems, which makes it easy to manage, and flexible to use. It is also backward compatible with WS-ReliableMessaging, which ensures reliable point-to-point Web services communications. This design enables the Web services to dynamically switch between replication and non-replication operation modes according to the availability requirement. Our framework has been carefully tuned to exhibit optimal performance, as shown in our measurement results. Future work includes the integration of our framework with the latest WS-Resource standard [13], and the support of transactional Web services. We are also planning to strengthen our framework to tolerate Byzantine faults [15].

References

[1] Apache Axiom, http://ws.apache.org/commons/axiom/.

[2] Apache Axis2 project, http://ws.apache.org/axis2/.

[3] Apache Sandesha2, http://ws.apache.org/sandesha/sandesha2/.

[4] R. Bilorusets et al., Web Services Reliable Messaging Protocol Specification, February 2005.

[5] K. Birman, Adding high availability and autonomic behavior to Web services, *Proceedings of the 26th International Conference on Software Engineering*, Scotland, UK, May 2004.

[6] K. Birman, Reliable Distributed Systems: Technologies, Web Services, and Applications, Springer, 2005.

[7] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, November 2002.

[8] P. Chan, M. Lyu, and M. Malek, "Making services fault tolerant," *Lecture Notes in Computer Science*, Vol. 4328, pp. 43–61, 2006.

[9] V. Dialani et al., "Transparent fault tolerance for Web services based architecture," *Lecture Notes in Computer Science*, Vol. 2400, pp. 889–898, 2002.

[10] G. Dobson, "Using WS-BPEL to implement software fault tolerance for Web services", *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 126-133, July 2006.

[11] A. Erradi, P. Maheshwari, "A broker-based approach for improving Web services reliability," *Proceedings of the IEEE International Conference on Web Services*, Orlando, Florida, July 2005.

[12] C. Fang et al., "Fault tolerant Web services," *Journal of Systems Architecture*, Vol.53, pp. 21–38, 2007.

[13] S. Graham et al., Web Services Resource 1.2, OASIS Standard, April 2006.

[14] JGroups: A Toolkit for Reliable Multicast Communication. http://www.jgroups.org.

[15] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, July 1982.

[16] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column)*, Vol. 32, No. 4 pp. 18–25, December 2001.

[17] N. Looker, M. Munro, J. Xu, "Increasing Web service dependability through consensus voting," *Proceedings of the 29th Annual International Computer Software and Applications Conference*, pp. 66-69, 2005.

[18] Object Management Group. Fault Tolerant CORBA (final adopted specification). OMG Technical Committee Document ptc/2000-04-04, April 2000.

[19] M. Merideth et al., "Thema: Byzantine-fault-tolerant middleware for Web services applications," *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pp. 131–142, 2005.

[20] J. Maurer, "A conversation with Roger Sessions and Terry Coatta," *ACM Queue*, Vol. 3, No. 7, pp. 16–25, 2005.

[21] L. Moser, M. Melliar-Smith, and W. Zhao, "Making Web services dependable," *Proceedings of the First International Conference on Availability, Reliability and Security*, Vienna University of Technology, Austria, pp. 440–448, April 2006,

[22] J. Salas et al., "WS-Replication: A framework for highly available Web services," *Proceedings of the 15th International Conference on World Wide Web*, Edinburgh, Scotland, pp. 357–366, May 2006.

[23] G. Santos, L. Lung, and C. Montez, "FTWeb: A fault tolerant infrastructure for Web services," *Proceedings of the IEEE International Enterprise Computing Conference*, Enschede, The Netherlands, pp. 95-105, September 2005.

[24] W. Vogels, "Web services are not distributed objects," *IEEE Internet Computing*, pp. 59–66, November-December, 2003.