BFT-WS: A Byzantine 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

Many Web services are expected to run with high degree of security and dependability. To achieve this goal, it is essential to use a Web-services compatible framework that tolerates not only crash faults, but Byzantine faults as well, due to the untrusted communication environment in which the Web services operate. In this paper, we describe the design and implementation of such a framework, called BFT-WS. BFT-WS is designed to operate on top of the standard SOAP messaging framework for maximum interoperability. It is implemented as a pluggable module within the Axis2 architecture, as such, it requires minimum changes to the Web applications. The core fault tolerance mechanisms used in BFT-WS are based on the well-known Castro and Liskov's BFT algorithm for optimal efficiency. Our performance measurements confirm that BFT-WS incurs only moderate runtime overhead considering the complexity of the mechanisms.

1. Introduction

Driven by business needs and the availability of the latest Web services technology, we have seen increasing reliance on services provided over the Web. We anticipate a strong demand for robust and practical fault tolerance middleware for such Web services. Considering the untrusted communication environment in which these services operate, arbitrary faults (crash faults as well as Byzantine faults [12]) must be tolerated to ensure maximum service dependability. Middleware that provides such type of fault tolerance is often termed as Byzantine fault tolerance (BFT) middleware.

There exist a well-known high quality research prototype [6] that provides Byzantine fault tolerance for generic client-server applications (similar prototypes are available, but they are often tied to a specific application, such as storage [18]). In fact, Merideth *et al.* [14] have used it directly for Web services fault tolerance. However, we argue against such an approach primarily for two reasons. First and foremost, the prototype uses proprietary messaging protocols (directly on top of IP multicast by default). This is incompatible with the design principles of Web services, which call for transport independence and mandate SOAP-based communications. The use of proprietary messaging protocols compromises the interoperability of Web services. Second, this prototype lacks direct support for Web services, which requires the use of a wrapper to mediate the two components. The mediation can be achieved either through an additional socket communication, which wastes precious system resources and is inefficient, or through a Java Native Interface (the vast majority of Web services are implemented in Java, and the BFT prototype [6] is implemented in C++), which is difficult to program and error-prone.

We believe that any type of middleware for Web services must use standard Web services technologies and must follow the design principles of Web services, and fault tolerance middleware for Web services is no exception. With this guideline in mind, we designed and implemented BFT-WS, a Byzantine fault tolerance framework for Web services. To avoid reinventing the wheel and to best utilize existing Web services technology, we decide to build BFT-WS by extending Sandesha2 [3], which is an implementation of the Web Service Reliable Messaging (WS-RM) standard [4] for Apache Axis2 [2] in Java. In BFT-WS, all fault tolerance mechanisms operate on top of the standard SOAP messaging framework for maximum interoperability. BFT-WS inherits Sandesha2's pluggability, and hence, it requires minimum changes to the Web applications (both the client and the service sides). The core fault tolerance mechanisms in BFT-WS are based on the well-known Castro and Liskov's BFT algorithm [6] for optimal runtime efficiency. The performance evaluation of a working prototype of BFT-WS shows that it indeed introduces only moderate runtime overhead verses the original Sandesha2 framework considering the complexity of the Byzantine fault tolerance mechanisms.

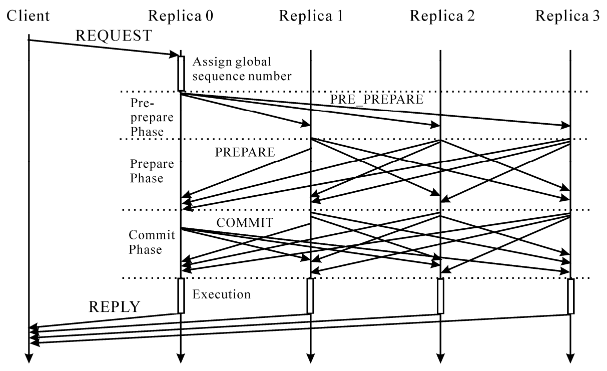

Figure 1. Normal operation of the BFT algorithm.

2. Background

2.1. Byzantine Fault Tolerance

A Byzantine faulty process may behave arbitrarily, in particular, it may disseminate different information to other processes, which constitutes a serious threat to the integrity of a system. Since a Byzantine faulty process may also choose not to respond to requests, it can exhibit crash fault behavior as well (*i.e.*, crash faults can be considered as a special case of Byzantine faults). Byzantine fault tolerance (BFT) refers to the capability of a system to tolerate Byzantine faults. For a client-server system, BFT can be achieved by replicating the server and by ensuring all server replicas to execute the same request in the same order. The latter means that the server replicas must reach an agreement on the set of requests and their relative ordering despite Byzantine faulty replicas and clients. Such an agreement is often referred to as Byzantine agreement [12].

Byzantine agreement algorithms had been too expensive to be practical until Castro and Liskov invented the BFT algorithm mentioned earlier [6]. The BFT algorithm is designed to support client-server applications running in an asynchronous distributed environment with a Byzantine fault model. The implementation of the algorithm contains two parts. At the client side, a lightweight library is responsible to send the client's request to the primary replica, to retransmit the request to all server replicas on the expiration of a retransmission timer (to deal with the primary faults), and to collect and vote on the replies. The main BFT algorithm is executed at the server side by a set of $3f+1$ replicas to tolerate f Byzantine faulty replicas. One of the replicas is designated as the primary while the rest are backups.

As shown in Figure 1, the normal operation of the (server-side) BFT algorithm involves three phases. During the first phase (called pre-prepare phase), the primary multicasts a pre-prepare message containing the client's request, the current view and a sequence number assigned to the request to all backups. A backup verifies the request message and the ordering information. If the backup accepts the message, it multicasts to all other replicas a prepare message containing the ordering information and the digest of the request being ordered. This starts the second phase, *i.e.*, the prepare phase. A replica waits until it has collected $2f$ prepare messages from different replicas (including the message it has sent if it is a backup) that match the pre-prepare message before it multicasts a commit message to other replicas, which starts the commit phase. The commit phase ends when a replica has received $2f$ matching commit messages from other replicas. At this point, the request message has been totally ordered and it is ready to be delivered to the server application if all previous requests have already been delivered. If the primary or the client is faulty, a Byzantine agreement on the ordering of a request might not be reached, in which case, a new view is initiated, triggered by a timeout on the current view. A different primary is designated in a round-robin fashion for each new view installed.

2.2. Web Services Reliable Messaging

The Web Services Reliable Messaging (WS-RM) standard describes a reliable messaging (RM) protocol between two endpoints, termed as RM source (RMS) and RM destination (RMD). The core concept introduced in WS-RM is *sequence*. A sequence is a unidirectional reliable channel between the RMS and the RMD. At the beginning of a reliable conversation between the two endpoints, a unique sequence (identified by a unique sequence ID) must first be created (through the create-sequence request and response). The sequence is terminated when the conversation is over (through the terminate-sequence request and response). For each message sent over the sequence, a unique message number must be assigned to it. The message number starts at 1 and is incremented by 1 for each subsequent message. The reliability of the messaging is achieved by the retransmission and positive acknowledgement mechanisms. At the RMS, a message sent is buffered and retransmitted until the corresponding acknowledgement from the RMD is received, or until a predefined retransmission limit has been exceeded. For efficiency reason, the RMD might not send acknowledgement immediately upon receiving an application message, and the acknowledgements for multiple messages can be piggybacked with another application message in the response sequence, or be aggregated in a single explicit acknowledgement message.

Because it is quite common for two endpoints to engage in two-way communications, the RMS can include an Offer element in its create-sequence request to avoid an explicit new sequence establishment step for the traffic in the reverse direction. Most interestingly, WS-RM defines a set of delivery assurances, including AtMostOnce, AtLeastOnce, Exactly-Once, and InOrder. The meaning of these assurances

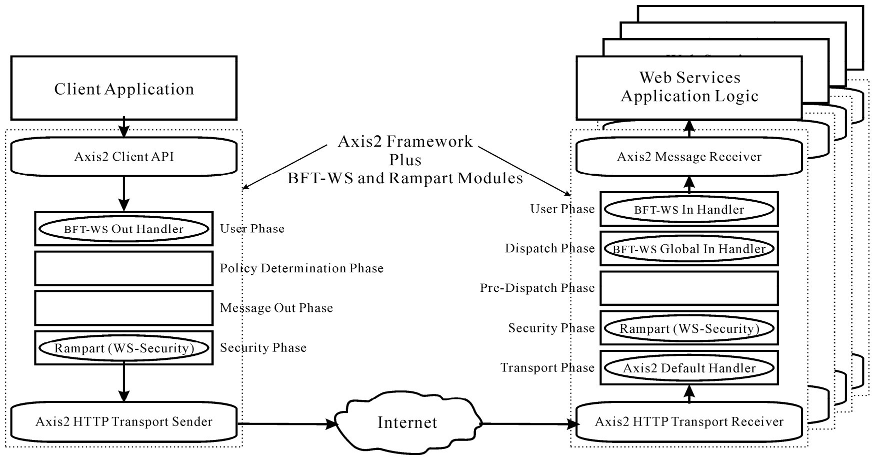

Figure 2. The overview of the BFT-WS architecture.

are self-explanatory. The InOrder assurance can be used together with any of the first three assurances. The strongest assurance is ExactlyOnce combined with InOrder delivery.

The WS-RM standard has been widely supported and there exist many implementations, most of which are commercial. We choose to use Sandesha2 [3] for this research, due to its open-source nature and its support for Axis2, the second generation open-source SOAP engine that supports pluggable modules.

3. BFT-WS System Architecture

The overview of the BFT-WS architecture is shown in Figure 2. BFT-WS is implemented as an Axis2 module. During the out-flow of a SOAP message, Axis2 invokes the BFT-WS Out Handler during the user phase, and invokes the Rampart (an Axis2 module that provides WS-Security [16] features) handler for message signing during the security phase. Then, the message is passed to the HTTP transport sender to send to the target endpoint. During the in-flow of a SOAP message, Axis2 first invokes the default handler for preliminary processing (to find the target object for the message based on the URI and SOAP action specified in the message) during the transport phase, it then invokes the Rampart handler for signature verification during the security phase. This is followed by the invocation of the BFT-WS Global In Handler during the dispatch phase. This handler performs tasks that should be done prior to dispatching, such as duplicate suppression at the server side. If the message is targeted toward a BFT-WS-enabled service, the BFT-WS In Handler is invoked for further processing during the user-defined phase, otherwise, the message is directly dispatched to the Axis2 message receiver. For clarity, Figure 2 shows only a one-way flow of a request from the client to the replicated Web service. The response flow is similar. Also not shown in Figure 2 are the multicast process and the internal components of the BFT-WS module.

Note that for the Rampart module to work (required by the BFT algorithm to authenticate the sender, so that a faulty replica cannot impersonate another correct replica), each replica has a pair of public and private RSA keys. Similarly, each client must also possess a public and private key pair. We assume that the public keys are known to all replicas and the clients, and the private keys of the correct replicas and clients are kept secret. We further assume the adversaries have limited computing power so that they cannot break the digital signatures of the messages sent by correct replicas or clients.

The main components of the BFT-WS module are illustrated in Figure 3. The client side bears a lot of similarity to the Sandesha2 client side module, with the exception of the addition of BFT-WS Voter, the replacement of Sandesha Sender by a Multicast Sender, and the replacement of the Sandesha Out Handler by the BFT-WS Out Handler. The server side contains more additions and modifications to the Sandesha2 components. Furthermore, a set of actions are added to the module configuration to allow total-ordering of messages, view change management and replica state synchronization. Besides the Multicast Sender, the server side introduced a Total Order Manager, and replaced the original Global In Handler, In Handler, and In-Order handler, by BFT-WS Global In Handler, BFT-WS In Handler and Total Order Invoker, respectively. The storage framework in Sandesha2 is not changed. The functions of these components (both Sandesha2 original and the modified or new components) are elaborated in the following subsections, starting with the components dealing with the out-flow, and then the components for the in-flow.

Note that even though what described in this section are specific to Axis2, we believe that our Byzantine fault

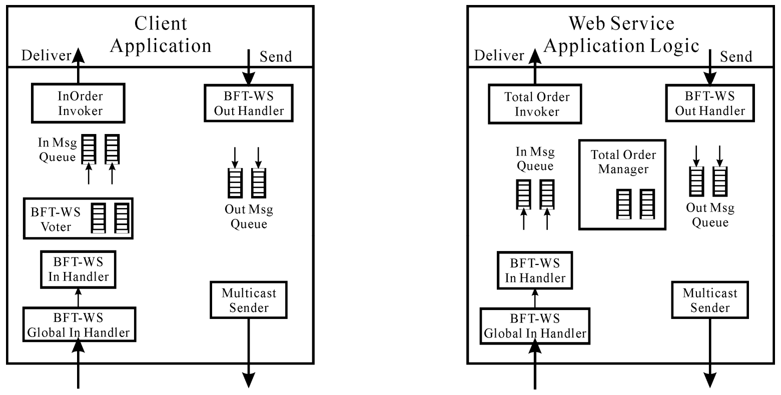

Figure 3. The main components of the BFT-WS module.

tolerance mechanisms are generic enough to be ported to other Web services infrastructure without great barrier.

3.1. BFT-WS Out Handler

This handler performs out-flow processing for reliable messaging. In particular, it generates a create-sequence request when the application sends the first message of a new sequence, and sends a terminate-sequence request after the last message of a sequence is transmitted. The difference between the BFT-WS Out Handler and the original Sandesha Out Handler lies in the creation and handling of the create-sequence message. In the original implementation, the create-sequence message does not contain any element that can be used for the server side to perform duplicate detection. If the create-sequence request contains an Offer element, it may be used as a way to check for duplicate. However, not all create-sequence requests contain such an element, because its existence is specified by the client application. To address this problem, we propose to include a UUID string in the create-sequence request. The UUID is embedded in the CreateSequence/any element, an optional element specified by the WS-RM standard to enable extensibility.

The addition of this UUID element also helps alleviate a tricky problem that would cause replica inconsistency. The WS-RM standard does not specify how the sequence ID for the newly created sequence should be determined. In Sandesha2, a UUID string is generated and used as the sequence ID at the server side. If we allow each replica to generate the sequence ID unilaterally in this fashion, the client would adopt the sequence ID present in the first create-sequence response it receives. This would prevent the client from communicating with other replicas, and would prevent the replicas from referring to the same sequence consistently when ordering the application messages sent over this sequence. Therefore, we modified the create-sequence request handling code to generate the sequence ID deterministically based on the client supplied UUID and the Web service group endpoint information.

3.2. Multicast Sender

In BFT-WS, the sequence between the client and the service provider endpoints is mapped transparently to a virtual sequence between the client and the group of replicas. The same sequence ID is used for the virtual sequence so that other components can keep referring to this sequence regardless if it is a one-to-one or a one-to-many (or many-to-one) sequence. The mapping is carried out by the multicast sender.

To make the mapping possible, we assume that each service to be replicated bears a unique group endpoint, in addition to the specific endpoint for each replica. Higher level components, including the application, must use the group endpoint when referring to the replicated Web service. When a message to the group endpoint is detected, including application messages and BFT-WS control messages, the multicast sender translates the group endpoint to a list of individual endpoints and multicasts the message to these endpoints. We assume the mapping information is provided by a configuration file. The Multicast Sender runs as a separate thread and periodically poll the Out Message Queue for messages to send.

One additional change is the garbage collection mechanism. For point-to-point reliable communication, it is sufficient to discard a buffered message as soon as an acknowledgement for the message is received. However, this mechanism does not work for reliable multicast for apparent reasons. Consequently, a message to be multicast is kept in the buffer until the acknowledgement from all destinations have been collected, or a predefined retransmission limit has been exceeded.

Note that in BFT-WS, the client multicasts its requests to all replicas via the Multicast Sender component. Even though it may be less efficient in some scenarios, such as when the client is geographically farther away from the Web service and the Web service replicas are close to each other, this design is more robust against adversary attacks since the clients do not need to know which replica is currently serving as the primary. Without such information, the adversary can only randomly pick up a replica to attack, instead of focusing on the primary directly. From the availability perspective, the compromise of the primary can result in much severe performance degradation than that of a backup. It is important to encapsulate internal state information as much as possible to improve system robustness. Information encapsulation also reduces the dependency between the clients and the Web services.

3.3. BFT-WS Global In Handler

The Sandesha Global In Handler performs duplicate filtering on application messages. This is fine for the server side, however, it would prevent the client from performing voting on the responses. Therefore, the related code is modified so that no duplicate detection is done on the client side. The other functionalities of this handler, *e.g.*, generating acknowledgement for the dropped messages, is not changed.

3.4. BFT-WS In Handler

Axis2 dispatches all application messages targeted to the BFT-WS-enabled services and the BFT-WS control messages to this handler. The BFT-WS In Handler operates differently for the client and the server sides.

At the client side, all application messages are passed immediately to the BFT-WS Voter component for processing. The rest of control messages are processed by the set of internal message processors as usual.

At the server side, all application messages are handled by an internal application message processor. Such messages are stored in the In Message Queue for ordering and delivery. All BFT-related control messages, such as pre-prepare, prepare, commit, and view change messages, are passed to the Total Order Manager for further processing. The WS-RM-related control messages such as create-sequence and terminate-sequence requests, are handled by the internal message processors available from the original Sandesha2 module, with the exception of the handling of sequence ID creation.

3.5. BFT-WS Voter

This component only exists at the client side. The Voter verifies the authenticity of the application messages received and temporarily stores the verified messages in its data structure. For each request issued, the Voter waits until it has collected $f + 1$ identical response messages from different replicas before it invokes the application message handler to process the response message. When the processing is finished, the message is passed to the In Message Queue for delivery.

3.6. Storage Manager

This component consists of the In Message Queue, the Out Message Queue, and a number of other subcomponents for sequence management, acknowledgement and retransmission management, and in-order delivery. This component comes with the Sandesha2 module. It is instrumented only for the purpose of performance profiling.

3.7. Total Order Invoker

This component replaces the Sandesha InOrder Invoker. This invoker runs as a separate thread to poll periodically the received application messages (stored in the In Message Queue) for ordering and delivery. To be eligible for ordering, the message must be in-order within its sequence, *i.e.*, all previous messages in the sequence has been received and ordered (or being ordered). If the message is eligible for ordering, the Total Order Manager is notified to order the message. Note that only the primary initiates the ordering of application messages.

The Total Order Invoker asks the Total Order Manager for the next message to be delivered. If there is a message ready for delivery, the Invoker retrieves the message from the In Message Queue and delivers it to the Web service application logic via the Axis2 message receiver.

3.8. Total Order Manager

This component is responsible for imposing a total order on all application requests according to the BFT algorithm. To facilitate reliable communication among the replicas themselves, each replica establishes a sequence with the rest of the replicas. The reliability of the control messages sent over these sequences are guaranteed by the WS-RM mechanisms and the Multicast Sender. For clarity, we first describe the BFT algorithm assuming that a unique global sequence number is assigned for each application message, then we elaborate on the batching mechanism which is needed to ensure optimal runtime performance. Due to space limitation, the view change and state transfer algorithms are omitted.

For each application request to be ordered, the Total Order Manager at the primary assigns the next available global sequence number to the message and constructs a pre-prepare message. The pre-prepare message contains the following information: The global sequence number n, the current view number v, and the digest d of the application message m to be ordered. The pre-prepare message is then passed to the Out Message Queue for sending.

The Total Order Manager uses a TotalOrderBean object to keep track of the ordering status for each application message. When a pre-prepare message is created at the primary, the Manager stores the message in the corresponding TotalOrderBean (a new TotalOrderBean is created on the creation of the first pre-prepare message for each application message).

At the backup, the Total Order Manager accepts a pre-prepare message if the message is signed properly, and it has not accepted a pre-prepare message for the same global sequence number *n* in view *v*. If the backup accepts the pre-prepare message, it creates a TotalOrderBean for the message and stores the pre-prepare message in the TotalOrderBean. The backup also constructs a prepare message containing the following information: The global sequence number *n*, the current view number *v*, the digest *d* of the application message *m*. The prepare message is dropped to the Out Message Queue for sending and the TotalOrderBean is updated correspondingly.

When a replica receives a prepare message, it verifies *n* and *v*, and compares the digest *d* with that of the application message. It accepts the prepare message if the check is passed and updates the TotalOrderBean. When a replica has collected 2*f* prepare messages from other replicas, including the pre-prepare message received from the primary (if the replica is a backup), it constructs a commit message with the same information as that of the prepare message, and passes the commit message to the Out Message Queue for sending. Again, the TotalOrderBean is updated for the sending of the commit message.

A replica verifies a commit message in the similar fashion to that for a prepare message. When a replica has collected 2*f* correct commit messages for *n* in view *v*, the message *m* is committed to the sequence number *n* in view *v*, *i.e.*, a total order for *m* has now been established. A totally ordered message can be delivered if all previously ordered messages have been delivered. We now describe the batching mechanism. The primary does not immediately order an application request when the message is in-order within its sequence 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 global sequence number for a group of application requests, at most one per sequence, and the requests ordered must be in-order within their own sequences.

4. Performance Evaluation

Our performance evaluation is carried out on a testbed consisting of 12 Dell SC440 servers connected by a 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 BFTWS framework during normal operation. A backup failure virtually does not affect the operation of the BFT 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 or delivered, due to the timeout value set 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 failures, the delay would be even longer.

An 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 message 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 throughput is measured at the replicated Web service. In our experiment, we keep the number of replicas to 4 (to tolerate a single Byzantine faulty replica), and vary the request sizes in terms of the number of elements in each request, and the number of concurrent clients.

Figure 4 shows the latency and throughput measurement results. In Figure 4(a), the end-to-end latency of the echo operation is reported for BFT replication with 4 replicas and a single client. For comparison, the latencies for two other configurations are also included. The first configuration involves no replication and no digital signing of messages. The second configuration involves no replication, but with all messages digitally signed. The measurements for the two configurations reveal the cost of digital signing and verification. As can be seen, such cost ranges from 90*ms* for short messages to 130*ms* for longer messages. The latency overhead of running BFT replication is significant. However, the overhead is very reasonable considering the complexity of the BFT algorithm. Comparing with the latency for the no-replication-with-signing configuration, the overhead ranges from 150*ms* for short messages to over 310*ms* for longer messages. The increased overhead for larger messages is likely due to the CPU contention for processing of the application requests (by the Web service) and the BFT replication mechanisms (by our framework). Future work is needed to fully characterize the sources of the additional cost for longer messages.

The latency cost for each step of processing in a request-reply round trip for a particular run with a single client and 4 replicas is summarized in Table 1. Both the request and the reply contain 1000 elements. As can be seen, the major costs come from message ordering, request multicasting, and message signing and verification.

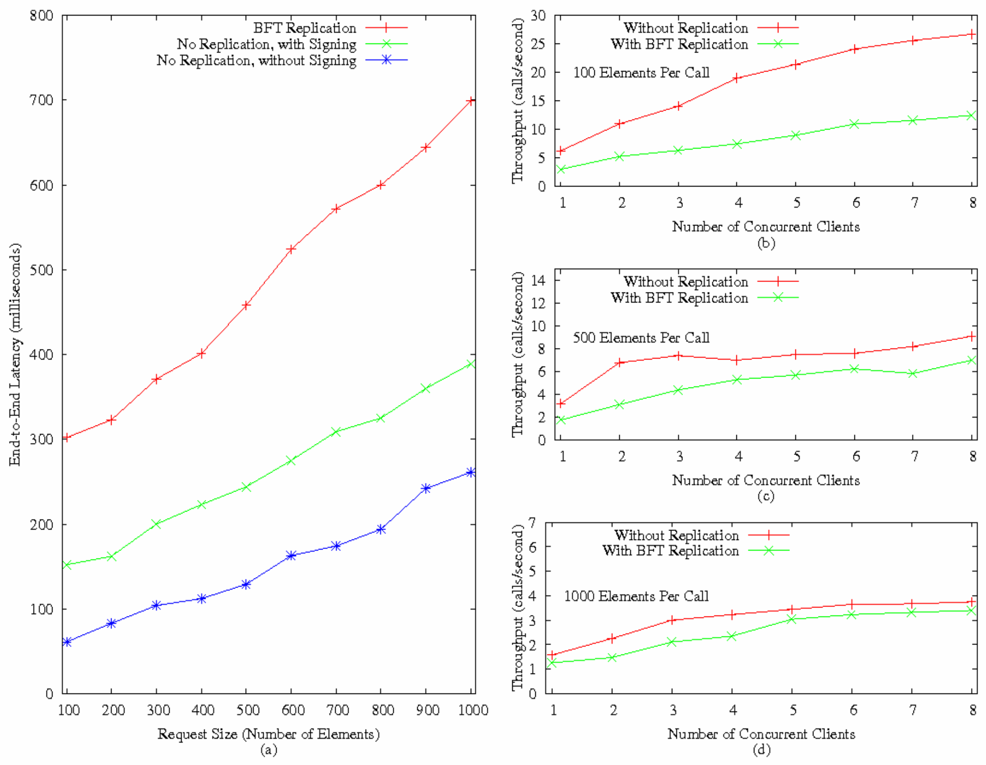

Figure 4. BFT-WS performance during normal operation. (a) The end-to-end latency. For comparison, the latency for the no-replication configuration with and without digital signing of messages are included as well. (b)-(d) Throughput *vs.* number of concurrent clients with different message sizes.

Table 1. Detailed latency measurement for a particular run with a single client and 4 replicas.

Processing Step	**Latency(ms)**
Request out-processing	9.7
Request multicast	127.5
Request in-processing	10.6
Request ordering and delivery	306.9
Request processing at application	43.6
Reply out-processing	5.4
Reply send	78.5
Reply voting	11.2
Reply in-processing	8.7
Reply delivery	16.3
Message signing & verification (derived)	130.0
Total	748.4

The throughput measurement results for different request sizes are shown in Figure 4(b) to (d). Note that the results for the no-replication configurations include digital signing and verification of all messages for fair comparison. It can be seen from Figure 4(b) that the throughput degradation is about 50% when BFT replication is enabled for short request sizes. Again, this is anticipated. Even with optimal batching for 8 concurrent clients, the primary must multicast 2 control messages (pre-prepare and commit) and receive 6 control messages (3 prepare and 3 commit messages from backups) to order the 8 application requests (recall that our measurements are carried out for normal operation with no replica failure). The approximately 50% reduction in throughput for short messages is nearly optimal. When the application request length and complexity is increased, the throughput reduction becomes far less, as shown in Figure 4(c) and (d).

5. Related Work

A large number of high availability solutions for Web services have been proposed in the last several years [5, 7-11, 13-15, 17]. Most of them are designed to cope with crash faults only. Furthermore, none of them has taken our approach, which integrates the replication mechanisms into the SOAP engine for maximum interoperability. Thema [14] is the only complete BFT framework for Web services which we know. In [17], an alternative solution is proposed for BFT Web services, but no implementation details or performance evaluations are reported.

Similar to our work, Thema [14] also relies on the BFT algorithm to ensure total ordering of application messages. However, a wrapper is used to interface with an existing implementation of the BFT algorithm [6], which is based on IP multicast, rather than the standard SOAP/HTTP transport, as such, it suffers from the interoperability problem we mentioned in the beginning of this paper. This approach limits its practicality. That said, it does provide richer functionality than our current BFT-WS framework in that it supports multi-tiered applications and the interaction between a replicated Web service as client and another non-replicated Web service as server. We plan to add similar functionality to BFT-WS in the next stage of our project.

[17] attempts to address some problems in Thema when a replicated client interacts with a replicated Web service which may have been compromised. It proposes to use the BFT algorithm for the client replicas to reach consensus on the reply messages to avoid the situation which different client replicas accept different reply messages for the same request made, when the server is compromised. However, it is not clear to us the value of such an approach. If the Web service has been compromised, the integrity of the service is no longer guaranteed, the service could easily send the *same* wrong reply to all client replicas, which cannot be addressed by the mechanism proposed in [17], and yet, the end-to-end latency is doubled as a result.

6. Conclusion and Future Work

In this paper, we presented the design and implementation of BFT-WS, a Byzantine fault tolerance middleware framework for Web services. It uses standard Web services technology to build the Byzantine fault tolerance service, and hence, it is more suitable to achieve interoperability. We also documented in detail the architecture and the major components of our framework. We anticipate that such descriptions are useful to practitioners as well as researchers working in the field of highly dependable Web services. Finally, our framework has been carefully tuned to exhibit optimal performance, as shown in our performance evaluation results. Future work will focus on the expansion of the feature set of BFT-WS, such as the support of multi-tiered Web services and transactional Web services.

Acknowledgement

This work was supported in part by Department of Energy Contract DE-FC26-06NT42853, and by a Faculty Research Development award from Cleveland State University. We also would like to thank the anonymous reviewers for their insightful comments.

References

[1] Apache Axiom. http://ws.apache.org/commons/axiom/.
[2] Apache Axis2. http://ws.apache.org/axis2 /index.html.
[3] Apache Sandesha2.
http://ws.apache.org/sandesha/sandesha2/index.html.
[4] R. Bilorusets et al. *Web Services Reliable Messaging Specification*, February 2005.
[5] K. Birman. "Adding high availability and autonomic behavior to web services". In *Proceedings of the International Conference on Software Engineering*, Scotland, UK, May 2004.
[6] M. Castro and B. Liskov. "Practical Byzantine fault tolerance". In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, USA, February 1999.
[7] P. Chan, M. Lyu, and M. Malek. "Making services fault tolerant". *Lecture Notes in Computer Science*, 4328:43–61, 2006.
[8] V. Dialani, S. Miles, L. Moreau, D. D. Roure, and M. Luck. "Transparent fault tolerance for web services based architecture". *Lecture Notes in Computer Science*, 2400:889–898, 2002.
[9] G. Dobson. "Using WS-BPEL to implement software fault tolerance for Web services". In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, July 2006.
[10] A. Erradi and P. Maheshwari. "A broker-based approach for improving Web services reliability". In *Proceedings of the IEEE International Conference on Web Services*, Orlando, FL, July 2005.
[11] C. Fang, D. Liang, F. Lin, and C. Lin. "Fault tolerant web services". *Journal of Systems Architecture*, 53:21–38, 2007.
[12] L. Lamport, R. Shostak, and M. Pease. "The Byzantine generals problem". *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
[13] N. Looker, M. Munro, and J. Xu. "Increasing web service dependability through consensus voting". In *Proceedings of the 29th Annual International Computer Software and Applications Conference*, pages 66–69, 2005.
[14] M. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. "Thema: Byzantine-fault-tolerant middleware for web services applications". In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 131–142, 2005.
[15] L. Moser, M. Melliar-Smith, and W. Zhao. "Making web services dependable". In *Proceedings of the 1st International Conference on Availability, Reliability and Security*, pages 440–448, Vienna University of Technology, Austria, April 2006.
[16] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *Web Services Security: SOAP Message Security 1.0*. OASIS, oasis standard 200401 edition, March 2004.
[17] S. L. Pallemulle, I. Wehrman, and K. J. Goldman. "Byzantine fault tolerant execution of long-running distributed applications". In *Proceedings of the IASTED Parallel and Distributed Computing and Systems Conference*, pages 528–523, November 2006.
[18] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. "Pond: the OceanStore prototype". In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, March 2003.