2007

Byzantine Fault Tolerant Coordination for Web Services Atomic Transactions

Wenbing Zhao
Cleveland State University, w.zhao1@csuohio.edu

Publisher's Statement
The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-540-74974-5_25

Original Citation
Zhao, W. (2007). Byzantine Fault Tolerant Coordination for Web Services Atomic Transactions. Lecture Notes in Computer Science, 4749, 307-318.

Byzantine Fault Tolerant Coordination for Web Services Atomic Transactions*

Wenbing Zhao

Department of Electrical and Computer Engineering
Cleveland State University, Cleveland, OH 44115
wenbing@ieee.org

Abstract. In this paper, we present the mechanisms needed for Byzantine fault tolerant coordination of Web services atomic transactions. The mechanisms have been incorporated into an open-source framework implementing the standard Web services atomic transactions specification. The core services of the framework, namely, the activation service, the registration service, the completion service, and the distributed commit service, are replicated and protected with our Byzantine fault tolerance mechanisms. Such a framework can be useful for many transactional Web services that require high degree of security and dependability.

Keywords: Reliable Service-Oriented Computing, Service-Oriented Middleware, Distributed Transactions, Byzantine Fault Tolerance.

1 Introduction

The bulk of business applications involve with transaction processing and require high degree of security and dependability. We have seen more and more such applications being deployed over the Internet, driven by the need for business integration and collaboration, and enabled by the latest service-oriented computing techniques such as Web services. This requires the development of a new generation of transaction processing (TP) monitors, not only due to the new computing paradigm, but because of the untrusted operating environment as well.

This work is an investigation of the issues and challenges of building a Byzantine fault tolerant (BFT) [1] TP monitor for Web services, which constitutes the major contribution of this paper. We focus on the Web services atomic transaction specification (WS-AT) [2]. The core services specified in WS-AT are replicated and protected with BFT mechanisms. The BFT algorithm in [3] is adapted for the replicas to achieve Byzantine agreement. We emphasize that the resulting BFT TP monitor framework is not a trivial integration of WS-AT and the BFT algorithm. As documented in detail in later sections, we proposed a number of novel mechanisms to achieve BFT with minimum overhead in the

* 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.

context of distributed transactions coordination, and the experimental evaluation of a working prototype proves the optimality of our mechanisms and their implementations.

2 Background

2.1 Byzantine Fault Tolerance

Byzantine fault tolerance refers to the capability of a system to tolerate Byzantine faults. It can be achieved by replicating the server and by ensuring all server replicas reach an agreement on the input despite Byzantine faulty replicas and clients. Such an agreement is often referred to as Byzantine agreement [1].

The most efficient Byzantine agreement algorithm reported so far is due to Castro and Liskov (referred to as the BFT algorithm) [3]. The BFT algorithm is executed 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. The normal operation of the 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 $2f$ matching prepare messages from different replicas before it multicasts a commit message to other replicas, which starts the third phase (*i.e.,* 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. To avoid possible confusion with the two phases (*i.e.,* the prepare phase and the commit/abort phase) in the two-phase commit (2PC) protocol [4], we refer the three phases in the BFT algorithm as ba-pre-prepare, ba-prepare, and ba-commit phases in this paper.

2.2 Web Services Atomic Transactions Specification

In WS-AT [2], a distributed transaction is modelled to have a coordinator, an initiator, and one or more participants. WS-AT specifies two protocol (*i.e.,* the 2PC protocol and the completion protocol), and a set of services. These protocols and services together ensure automatic activation, registration, propagation, and atomic termination of Web-services based distributed transactions. The 2PC protocol is run between the coordinator and the participants, and the completion protocol is run between the initiator and the completion service. The initiator is responsible to start and end a transaction. The coordinator side consists of the following services:

– *Activation Service*: It is responsible to create a coordinator object (to handle registration, completion, and distributed commit) and a transaction context for each transaction.
– *Registration Service*: It is provided to the transaction participants and the initiator to register their endpoint references for the associated participant-side services.
– *Coordinator Service*: This service is responsible to run the 2PC protocol to ensure atomic commitment of a distributed transaction.
– *Completion Service*: This service is used by the transaction initiator to signal the start of a distributed commit.

The set of coordinator services run in the same address space. For each transaction, all but the Activation Service are provided by a (distinct) coordinator object. The participant-side services include:

– *CompletionInitiator Service*: It is used by the coordinator to inform the transaction initiator the final outcome of the transaction, as part of the completion protocol.
– *Participant Service*: The coordinator uses this service to solicit votes from, and to send the transaction outcome to the participants.

The detailed steps of a distributed transaction using a WS-AT conformant framework are shown with a banking example (adapted from [5] and used in our performance evaluation) in Fig. 1. In this example, a bank provides an online banking Web service that a customer can access. The transaction is started due to a single Web service call from the customer on the bank to transfer some amount of money from one account to the other.

3 System Models

We consider a composite Web service that utilizes Web services provided by other departments or organizations, similar to the example shown in Fig. 1. We assume that an end user uses the composite Web service through a Web browser or directly invokes the Web service interface through a standalone client application. In response to each request from an end user, a distributed transaction is started to coordinate the interactions with other Web services. The distributed transactions are supported by a WS-AT conformant framework such as [5].

For simplicity, we assume a flat distributed transaction model. We assume that for each transaction, a distinct coordinator is created. The lifespan of the coordinator is the same as the transaction it coordinates.

The composite Web service provider serves as the role of the initiator. We assume that the initiator is stateless because it typically provides only a front-end service for its clients and delegates actually work to the participants. All transactions are started and terminated by the initiator. The initiator also propagates the transaction to other participants through a transaction context included in the requests.

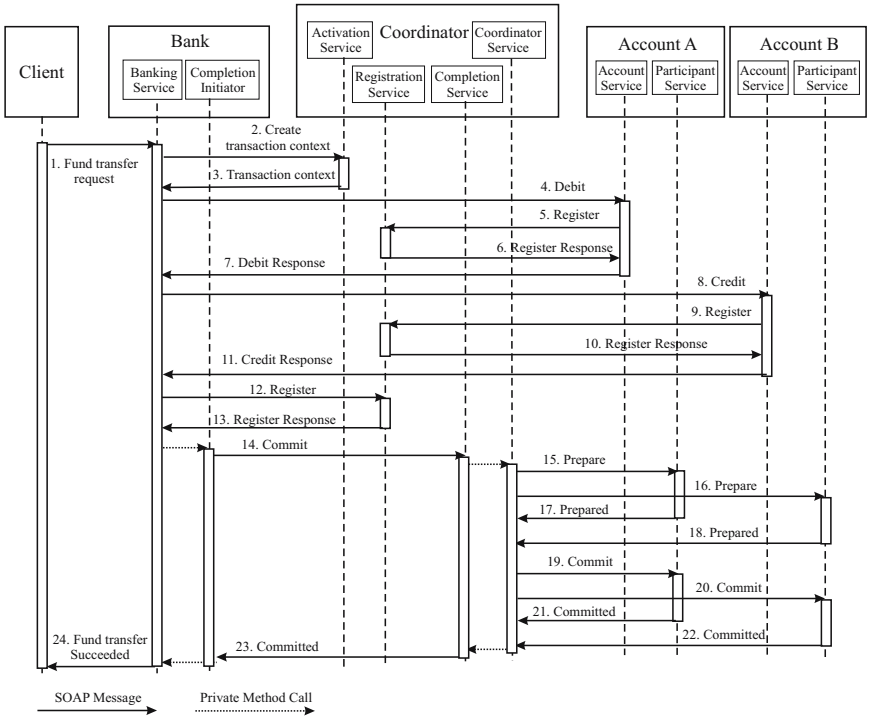

Fig. 1. The sequence diagram of a banking example using WS-AT

We assume that the transaction coordinator runs separately from the initiator and the participants.[1] Both the coordinator and the initiator are replicated. For simplicity, we assume that the participants are not replicated. We assume that $3f + 1$ coordinator replicas are available, among which at most f can be faulty during a transaction. Because the initiator is stateless, we require only $2f + 1$ initiator replicas to tolerate f faulty initiator replicas. There is no limit on the number of faulty participants.

We call a coordinator/initiator replica correct if it does not fail during its lifetime, *i.e.,* it faithfully executes according to the protocol prescribed from the start to the end. However, we call a participant correct if it is not Byzantine faulty, *i.e.,* it may be subject to typical non-malicious faults such as crash faults or performance faults.

[1] Even though it is a common practice to collocate the initiator with the coordinator in the same node, it might not be a desirable approach, due to primarily two reasons. First, collocating the initiator and the coordinator tightly couples the business logic with the generic transaction coordination mechanism (also observed in [6]), which is desirable neither from the software engineering perspective (it is harder to test) nor from the security perspective (it is against the defence-in-depth principle). Second, the initiator typically is stateless, which can be rendered fault tolerant fairly easily, while the coordination service is stateful. This naturally calls for the separation of the initiator and the coordinator.

The coordinator and initiator replicas are subject to Byzantine faults, *i.e.,* a Byzantine faulty replica can fail arbitrarily. For participants, however, we have to rule out some forms of Byzantine faulty behaviors. A Byzantine faulty participant can always vote to abort, or it can vote to commit a transaction, but actually abort the transaction locally. It is beyond the scope of any distributed commit protocol to deal with these situations. Rather, they should be addressed by business accountability and non-repudiation techniques. Other forms of participant faults, such as a faulty participant sending conflicting votes to different coordinator replicas, will be tolerated.

All messages between the coordinator and the participants are digitally signed. We assume that the coordinator replicas and the participants each has a public/secret key pair. The public keys of the participants are known to all coordinator replicas, and vice versa, while the private key is kept secret to its owner. We assume that the adversaries have limited computing power so that they cannot break the encryption and digital signatures of correct coordinator replicas.

4 Byzantine Fault Tolerance Mechanisms

4.1 Activation

Figure 2 shows the mechanisms for the activation process. Upon receiving a request from a client, the initiator starts a distributed transaction and sends an activation request to the activation service. The client's request has the form $<\text{CREQ}, o, t, c>_{\sigma_c}$, where o is the operation to be executed by the initiator, t is a monotonically increasing timestamp, c is the client id, and σ_c is the client's digital signature for the request. A correct client sends the request to all initiator replicas. An initiator accepts the request if it is properly signed by the client, and it has not accepted a request with equal or larger timestamp from the same client. If the request carries an obsolete timestamp, the cached reply is retransmitted if one is found in the reply log.

The activation request has the form $<\text{ACTIVATION}, v, c, t, k>_{\sigma_k}$, where v is the current view, k is the initiator replica id. The request is sent to the primary replica of the activation service. The primary initially logs the activation request if the message is correctly signed by the initiator replica and it has not accepted a request with equal or larger timestamp from the initiator in view v. Only when $f + 1$ such messages are received from different initiator replicas with matching c and t, does the primary accept the activation request. This is to ensure the request comes from a correct initiator replica. The primary then sends a ba-pre-prepare message to the backup replicas. The ba-prepare message has the form $<\text{BA-PRE-PREPARE}, v, r, uuid_p, p>_{\sigma_p}$, where r is the content of the activation request, p is the primary id, $uuid_p$ is a universally unique identifier (UUID) proposed by the primary.

The UUID is used to generate the transaction id, which will be used to identify the transaction and its coordinator object. To maximize security, the UUID should be generated from a high entropy source, which means the activation operation is inherently nondeterministic, and the UUID proposed by one replica cannot be

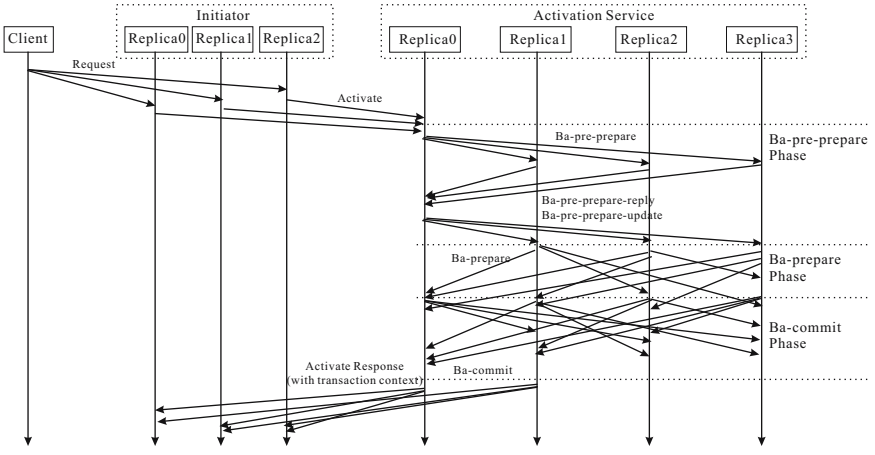

Fig. 2. Byzantine fault tolerance mechanisms for the activation of a transaction

verified by another. This calls for the collective determination of the UUID for the transaction.[2] This is achieved during the ba-pre-prepare phase.

A backup activation replica accepts the ba-prepare message if it is in view v, the message is properly signed, r is a correct activation request, and it has not accepted the same message before. The backup then sends a ba-pre-prepare-reply message in the form $<$BA-PRE-PREPARE-REPLY$, v, d, uuid_i, i>_{\sigma_i}$ to all replicas, where d is the digest of the ba-prepare message, i is the replica id and $uuid_i$ is i's UUID proposal. When the primary collects $2f$ ba-prepare-reply messages from different backups, it sends a ba-prepare-update message in the form $<$BA-PRE-PREPARE-UPDATE$, v, d, U, p>_{\sigma_p}$ to the backup replicas, where U is the collection of the digests of the $2f$ ba-pre-prepare-reply messages.

A backup accepts a ba-pre-prepare-reply message if it is in view v, the message is properly signed and d matches the digest of the ba-pre-prepare message. It accepts the ba-pre-prepare-update message if it is in view v, d matches that of the ba-pre-prepare message, and the digests in U match that of the ba-pre-prepare-reply messages. It is possible that a backup has not received a particular ba-pre-prepare-reply message, in which case, the backup asks for a retransmission from the primary. Upon accepting the ba-pre-prepare-update message, a backup sends a ba-prepare message to all replicas. The message has the form $<$BA-PREPARE$, v, d, uuid, i>_{\sigma_i}$, where $uuid$ is the final UUID computed deterministically based on the proposals from the primary and $2f$ backups (we choose to use the average of the group of UUIDs as the final UUID, but other computation method is possible). A replica accepts a ba-prepare message if it is in view v, the

[2] One might attempt to replace the high entropy source with a deterministic source to ensure the replica consistency. However, doing so might result in an easy-to-predict transaction identifier, which opens the door for replay attacks. An alternative to our approach is the coin-tossing scheme [7], however, it requires an additional phase to securely distribute the private key shares to the replicas.

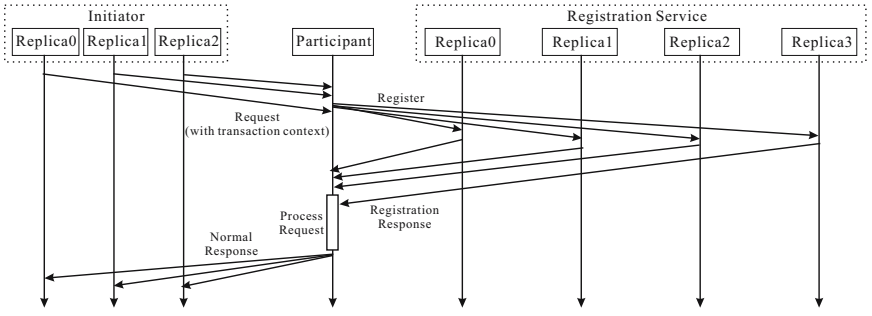

Fig. 3. Byzantine fault tolerance mechanisms for the registration of a participant

message is properly signed, d is the digest of the ba-pre-prepare message, and *uuid* matches its own.

When an activation replica has accepted $2f$ ba-prepare messages from different replicas (including the message it sent), in addition to the ba-pre-prepare and ba-pre-prepare-update messages it has accepted or has sent (if it is the primary), it sends a ba-commit message in the form $<\text{BA-COMMIT}, v, d, uuid, i>_{\sigma_i}$ to all other replicas. The verification of the ba-commit message is similar to that of the ba-prepare message. When a replica accepts $2f + 1$ matching ba-commit messages from different replicas (including the message it has sent), it calculates (deterministically) the transaction id *tid* based on *uuid*, creates a coordinator object with the *tid*, and sends the activation response to the initiator replicas. The response has the form $<\text{ACTIVATION-RESPONSE}, c, t, C, i>_{\sigma_i}$, where c and t are the client id and the timestamp included in the activation request, C is the transaction context. Note that if the primary is faulty, it can prevent a correct replica from completing the three phases, in which case, the replica suspects the primary and initiates a view change.

An initiator replica logs the activation response if it is properly signed, and c and t match those in its activation request. The replica accepts the message if it has collected $f + 1$ matching responses from different activation service replicas.

4.2 Registration and Transaction Propagation

To ensure atomic termination of a distributed transaction, it is essential that all correct coordinator replicas agree on the set of participants involved in the transaction. This can be achieved by running a Byzantine agreement algorithm among the coordinator replicas whenever a participant registers itself. However, doing so might incur too much overhead for the coordination service to be practical. In this work, we defer the Byzantine agreement on the participants set until the distributed commit stage and combine it with that for the transaction outcome. This optimization is made possible by the mechanisms shown in Fig. 3. In addition, we assume that there is proper authentication mechanism in place to prevent a Byzantine faulty process from illegally registering itself as a participant at correct coordinator replicas.

A participant does not accept a request until it has collected $f + 1$ matching requests from different initiator replicas. This is to prevent a faulty initiator replica from excluding a participant from joining the transaction (*e.g.,* by not including the transaction context in the request), or from including a process that should not participate the transaction. Since at most f initiator replicas are faulty, one of the messages must have been sent by a correct initiator replica.

To register, a participant sends its registration request to all coordinator replicas and waits until $2f + 1$ acknowledgments from different replicas have been collected. Since at most f replicas are faulty, at least $f + 1$ correct replicas must have accepted the registration request. If the participant can register successfully and complete its execution of the initiator's request, it sends a normal reply to the initiator replicas. Otherwise, it sends an exception back (possibly after recovery from a transient failure). If an initiator replica receives an exception from a participant, or times out a participant, it should choose to abort the transaction.

The initiator replicas also register with the coordinator replicas prior to the termination of the transaction. It follows a similar mechanism as that of the participants. Because at most f initiator replicas are faulty, at least $f+1$ replicas can finish the registration successfully.

4.3 Completion and Distributed Commit

The Byzantine fault tolerant transaction completion and distributed commit mechanisms are illustrated in Fig. 4. When an initiator replica completes all the operations successfully within a transaction, it sends a commit request to the coordinator replicas. Otherwise, it sends a rollback request. A coordinator replica does not accept the commit or rollback request until it has received $f + 1$ matching requests from different initiator replicas.

Upon accepting a commit request, a coordinator replica starts the first phase of the standard 2PC protocol. However, at the end of the first phase, a Byzantine agreement phase is conducted so that all correct coordinator replicas agree on the same outcome and the participants set for the transaction. This will be followed by the second phase of the 2PC protocol. If a rollback request is received, the first phase of 2PC is skipped, but the Byzantine agreement phase is still needed before the final decision is sent to all participants. When the distributed commit is completed, the coordinator replicas inform the transaction outcome to the initiator replicas. An initiator replica accepts such a notification only if it has collected $f + 1$ matching messages from different coordinator replicas. Similarly, a participant accepts a prepare request, or a commit/rollback notification only if it has collected $f + 1$ matching messages for the same transaction from different coordinator replicas. Again, this is to ensure the request or notification comes from a correct replica.

As shown in Fig. 4, the Byzantine agreement algorithm used for distributed commit is similar to that in Sect. 4.1, except that no ba-pre-prepare-reply and ba-pre-prepare-update messages are involved and the content of the messages are different. Due to space limitation, we only describe the format and the verification criteria for each type of messages used.

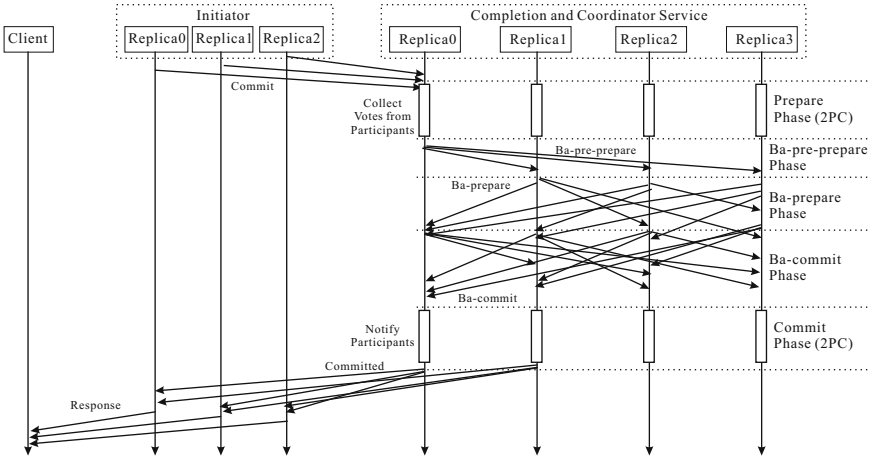

Fig. 4. Byzantine fault tolerance mechanisms for completion and distributed commit

The ba-pre-prepare message has the form $<\text{BA-PRE-PREPARE}, v, tid, o, C>_{\sigma_p}$, where o is the proposed transaction outcome (*i.e.,* commit or abort), C is the decision certificate, and σ_p is the primary's signature for the message. The decision certificate contains a collection of records, one for each participant. The record for a participant j contains a signed registration $R_j = (tid, j)_{\sigma_j}$ and a signed vote $V_j = (tid, vote)_{\sigma_j}$ if a vote from j has been received by the primary. The *tid* is included in each registration and vote record so that a faulty primary cannot reuse an obsolete registration or vote record to force a transaction outcome against the will of some correct participants.

A backup accepts a ba-pre-prepare message provided it is in view v, it is handling *tid*, the message is signed properly, the registration records in C are identical to, or form a superset of, the local registration records, and it has not accepted another ba-pre-prepare message for *tid* in view v. It also verifies that every vote record in C is properly signed by its sending participant and the *tid* in the record matches that of the current transaction, and the proposed decision o is consistent with the registration and vote records.

The ba-prepare message takes the form $<\text{BA-PREPARE}, v, t, d, o, i>_{\sigma_i}$, where d is the digest of the decision certificate C. A coordinator replica accepts a ba-prepare message provided it is in view v, it is handling *tid*, the message is correctly signed by replica i, the decision o matches that in the ba-pre-prepare message, and the digest d matches that of the decision certificate in the accepted ba-pre-prepare message.

The ba-commit message has the form $<\text{BA-COMMIT}, v, tid, d, o, i>_{\sigma_i}$. The ba-commit message is verified using the same criteria as those for ba-prepare messages. When a replica collects $2f + 1$ matching ba-commit messages from different replicas, it sends the decision o to all participants of transaction *tid*. If a replica i could not reach an agreement, it initiates a view change when a timeout occurs.

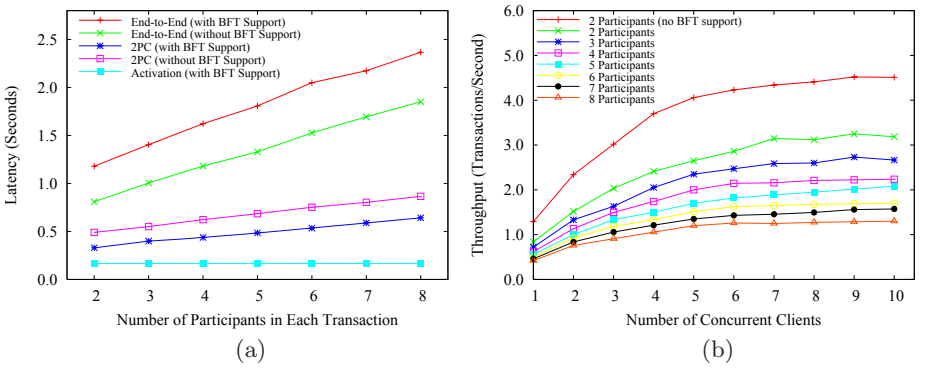

Fig. 5. (a) Various latency measurements under normal operations (with a single client). (b) End-to-end throughput under normal operations.

5 Implementation and Performance Evaluation

We have implemented the core Byzantine fault tolerance mechanisms (with the exception of the view change mechanisms) and integrated them into Kandula [5], a Java-based open source implementation of WS-AT. The extended framework also uses WSS4J (an implementation of the Web Services Security Specification) [8], and Apache Axis (SOAP Engine) 1.1 [9]. Due to space limitation, the implementation details are omitted.

Our experiment is carried out on a testbed consisting of 20 Dell SC1420 servers connected by a 100Mbps Ethernet. Each server is equipped with two Intel Xeon 2.8GHz processors and 1GB memory running SuSE 10.2 Linux.

The test application is the banking Web services application described in Sec. 2.2. The initiator is replicated across 3 nodes, and the coordination services are replicated on 4 nodes. The participants and the clients are not replicated, and are distributed among the remaining nodes. Each client invokes a fund transfer operation on the banking Web service within a loop without any "think" time between two consecutive calls. In each run, 1000 samples are obtained. The end-to-end latency for the fund transfer operation is measured at the client. The latency for the transaction activation and distributed commit are measured at the coordinator replicas. Finally, the throughput of the distributed commit service is measured at the initiator for various number of participants and concurrent clients.

As can be seen in Fig. 5(a), the end-to-end latency for a transaction is increased by about 400-500 ms when the number of participants varies from 2 to 8. The increase is primary due to the two Byzantine agreement phases in our mechanisms (one for activation, the other for 2PC). The latencies for transaction activation and for 2PC are also shown in Fig. 5(a). While the latency for 2PC increases with the number of participants, the activation latency remains constant because the participants are not involved with activation. As shown in Fig. 5(b), the throughput for transactions using our mechanisms is

about 30% to 40% lower than those without replication protection, which is quite moderate considering the complexity of the BFT mechanisms. (To avoid cluttering, only the 2-participants case is shown for the no-replication configuration.)

6 Related Work

There are a number of system-level work on fault tolerant TP monitors, such as [10,11]. However, they all use a benign fault model. Such systems do not work if the coordinator is subject to intrusion attacks. We have yet to see other system-level work on Byzantine fault tolerant TP monitors. The work closest to ours is Thema [12], which is a BFT framework for generic multi-tiered Web services. Even though some of the mechanisms are identical, our work contains specific mechanisms to ensure atomic transaction commitment.

The problem of BFT distributed commit for atomic transactions has been of research interest in the past two decades [13,14]. The first such protocol is proposed by Mohan et al. [13]. In [13], the 2PC protocol is enhanced with a Byzantine agreement phase on the transaction outcome among the coordinator and all participants in the root cluster. This approach has several limitations. First, the atomicity of a transaction is guaranteed only for participants residing in the root cluster under Byzantine faults. Second, it requires every participant within the cluster knows the cluster membership, which may not be applicable to Web services atomic transactions because a participant is not obligated to know all other participants. Our work, on the other hand, requires a Byzantine agreement only among the coordinator replicas and hence, allows dynamic propagation of transactions. Rothermel *et al.* [14] addressed the challenges of ensuring atomic distributed commit in open systems where participants may be compromised. However, [14] assumes that the root coordinator is trusted. This assumption negates the necessity to replicate the coordinator for Byzantine fault tolerance. Apparently, this assumption is not applicable to Web services applications.

7 Conclusion and Future Work

In this paper, we presented Byzantine fault tolerance mechanisms for distributed coordination of Web services atomic transactions. We focus on the protection of the basic services and infrastructures provided by typical TP monitors against Byzantine faults. By exploiting the semantics of the distributed coordination services, we are able to adapt Castro and Liskov's BFT algorithm [3] to ensure Byzantine agreement on the transaction identifiers and the outcome of transactions fairly efficiently. A working prototype is built on top of an open source distributed coordination framework for Web services. The measurement results show only moderate runtime overhead considering the complexity of Byzantine fault tolerance. We believe that our work is an important step towards a highly

secure and dependable TP monitor for Web services.[3] We are currently working on the implementation of the view change mechanisms and conducting experiments in the wide-area network configurations.

Acknowledgement. We wish to thank the anonymous reviewers for their insightful comments on an earlier draft of this paper.

References

1. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)
2. Cabrera, L., et al.: WS-AtomicTransaction Specification (August 2005)
3. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems 20(4), 398–461 (2002)
4. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, CA (1983)
5. Apache Kandula project, `http://ws.apache.org/kandula/`
6. Erven, H., Hicker, H., Huemer, C., Zapletal, M.: Web Services-BusinessActivity-Initiator (WS-BA-I) Protocol: an extension to the Web Services-BusinessActivity specification. In: Proceedings of the IEEE Internaltion Conference on Web Services, Salt Lake City, Utah (July 2007)
7. Cachin, C., Kursawe, K., Shoup, V.: Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In: Proceedings of the 19th ACM Symposium on Principles of Distributed Computing, pp. 123–132. ACM Press, New York (2000)
8. Apache WSS4J project, `http://ws.apache.org/wss4j/`
9. Apache Axis project, `http://ws.apache.org/axis/`
10. Frolund, S., Guerraoui, R.: e-Transactions: End-to-end reliability for three-tier architectures. IEEE Transactions on Software Engineering 28(4), 378–395 (2002)
11. Zhao, W., Moser, L.E., Melliar-Smith, P.M.: Unification of transactions and replication in three-tier architectures based on CORBA. IEEE Transactions on Dependable and Secure Computing 2(2), 20–33 (2005)
12. Merideth, M., Iyengar, A., Mikalsen, T., Tai, S., Rouvellou, I., Narasimhan, P.: Thema: Byzantine-fault-tolerant middleware for web services applications. In: Proceedings of the IEEE Symposium on Reliable Distributed Systems, pp. 131–142. IEEE Computer Society Press, Los Alamitos (2005)
13. Mohan, C., Strong, R., Finkelstein, S.: Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. In: Proceedings of the ACM symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, pp. 89–103. ACM Press, New York (1983)
14. Rothermel, K., Pappe, S.: Open commit protocols tolerating commission failures. ACM Transactions on Database Systems 18(2), 289–332 (1993)

[3] In the current stage, due to the high redundancy level required and the high degree of complexity imposed by the BFT mechanisms, the solutions proposed in this paper are useful only for those applications that are so mission critical that the cost of doing so is well justified.