Trustworthy Coordination of Web Services Atomic Transactions

Honglei Zhang, Hua Chai, Wenbing Zhao, *Member, IEEE*,
P. M. Melliar-Smith, *Member, IEEE*, L. E. Moser, *Member, IEEE*

***Abstract*—The Web Service Atomic Transactions (WS-AT) specification makes it possible for businesses to engage in standard distributed transaction processing over the Internet using Web Services technology. For such business applications, trustworthy coordination of WS-AT is crucial. In this paper, we explain how to render WS-AT coordination trustworthy by applying Byzantine fault tolerance (BFT) techniques. More specifically, we show how to protect the core services described in the WS-AT specification, namely, the Activation service, the Registration service, the Completion service and the Coordinator service, against Byzantine faults. The main contribution of this work is that it exploits the semantics of the WS-AT services to minimize the use of Byzantine agreement, instead of applying BFT techniques naively, which would be prohibitively expensive. We have incorporated our BFT protocols and mechanisms into an open-source framework that implements the WS-AT specification. The resulting BFT framework for WS-AT is useful for business applications that are based on WS-AT and that require a high degree of dependability, security and trust.**

***Index Terms*—Atomic transactions, distributed transactions, service oriented computing, Web Services, dependability, security, trust, encryption, authentication, Byzantine fault tolerance**

I. Introduction

Driven by the need for business collaboration and integration, more and more applications are being deployed over the Internet using Web Services technology. Many such applications involve distributed transaction processing. To provide interoperability among transactional Web Services, the Web Services Atomic Transactions (WS-AT) specification [18] was developed by a consortium of companies (led by Microsoft and IBM) and was recently adopted by OASIS as one of the Web Services standards.

According to the WS-AT specification, the Coordinator offers a set of services to the Initiator and the Participants of a transaction, namely, the Activation service, the Registration service, the Completion service, and the Coordinator service. The Activation service creates a Coordinator object and a transaction context for each new transaction. The Registration service admits a Participant into the transaction. The Completion service initiates the distributed commit of the transaction at the request of the Initiator. The Coordinator service coordinates the Participants to commit or abort the transaction atomically.

H. Zhang, H. Chai, and W. Zhao are with the Department of Electrical and Computer Engineering, Cleveland State University, Cleveland, OH 44115.

P. M. Melliar-Smith and L. E. Moser are with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106.

The objective of the research presented in this paper is to harden these WS-AT services so that they are trustworthy, even in the presence of Byzantine faults [17], within the untrusted environment of the Internet.

One might expect that the WS-AT services could be easily rendered Byzantine fault tolerant by replicating the WS-AT services and ensuring Byzantine agreement of the replicas on every operation using an existing BFT algorithm, such as the Practical Byzantine Fault Tolerance (PBFT) algorithm [9]. However, such an approach is not practical, because executing Byzantine agreement on every operation is prohibitively expensive, as our experimental results in Section V show.

The main contribution of this paper is a lightweight BFT framework for trustworthy coordination of Web Services Atomic Transactions that exploits the semantics of the WS-AT interactions to achieve better performance than a general-purpose BFT algorithm that is naively applied. We recognize that not every operation in WS-AT requires Byzantine agreement among the Coordinator replicas and, thus, that the total number of Byzantine agreements needed in a typical transaction can be sharply reduced.

More specifically, our BFT framework uses a lightweight protocol instead of running an instance of Byzantine agreement for registration of each Participant. The protocol utilizes, at each Participant, the collection of registration acknowledgments from a quorum of Coordinator replicas, and a round of message exchange at the start of the two-phase commit protocol. These mechanisms ensure that, if a non-faulty Participant has registered with the Coordinator, the Participant is included in the two-phase commit. Moreover, if the number of Participants is large, these mechanisms reduce the overhead dramatically.

In addition, our BFT framework introduces a decision certificate into the transaction outcome notification during the two-phase commit, to limit the malicious impact of a (not yet detected) Byzantine faulty replica on the Participants (and other replicas). The decision certificate makes it possible to run a single instance of Byzantine agreement on the decision of the transaction outcome during two-phase commit, instead of running an instance of Byzantine agreement for each Participant on the vote of that Participant. For transactions with a large number of Participants, the decision certificate improves the performance significantly. Furthermore, because the decision certificate includes the signed votes of the Participants, it enables accountability. For example, if a faulty Participant votes to commit a transaction, but aborts the transaction locally, other Participants can hold it accountable, because of the signed votes they have.

For convenience, we choose to use the PBFT algorithm [9] for Byzantine agreement in the implementation of our BFT framework. Other BFT algorithms [1], [2], [7], [16], [33] potentially offer better performance during normal operation and/or more robustness under various kinds of attacks. Most of these algorithms could be used in our BFT framework instead of PBFT, without significant changes other than the implementation of the BFT algorithm itself.

Finally, our work focuses on trustworthy coordination of distributed transactions for Web Services. It is not our objective to provide a Byzantine fault tolerance framework for database systems, as [27], [32] do.

II. Background

A. Web Services

The Web Services technologies comprise a set of standards that enable automated machine-to-machine interactions over the Internet. The key Web Services technologies are the eXtensible Markup Language (XML) [6], the Simple Object Access Protocol (SOAP) [14], and the Web Services Description Language (WSDL) [11].

XML is designed to facilitate self-contained, structured data representation and transfer over the Internet. The extensibility of XML makes it the essential building block for Web Services. Extensibility refers to the ability to introduce additional features and functionalities. XML is extensible in that it allows users to define their own tags.

SOAP can be used either to conduct remote procedure calls or to exchange XML documents over the Internet. A SOAP message contains a SOAP Envelope and a SOAP Body. A SOAP message often contains an optional SOAP Header element, and sometimes contains a Fault element if an error occurs. Typically, HTTP is used to transport SOAP messages over the Internet.

WSDL is an XML-based language used to describe Web Services. For each Web Service, the corresponding WSDL document specifies the available operations, the relevant messages, and a set of endpoints to reach the Web Service. Due to its use of XML, WSDL is also extensible.

B. Web Services Atomic Transactions Specification

The Web Service Atomic Transactions (WS-AT) standard specifies two protocols (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 a distributed transaction based on Web Services. The 2PC protocol [34] is run between the Participants and the Coordinator, and the Completion protocol is run between the Coordinator and the Initiator. The Initiator is responsible for not only starting the distributed transaction but also ending it.

The Participants register with the Coordinator when they become involved in the transaction. The 2PC protocol commits a transaction in two phases. During the first phase (the Prepare phase), the Coordinator disseminates a request to all of the Participants so that they can prepare to commit the transaction. If a Participant is able to commit the transaction, it prepares the transaction for commitment and responds with a Prepared vote; otherwise, the Participant responds with an Abort vote. When a Participant responds with a Prepared vote, it enters the Ready state. A Participant must be prepared to either commit or abort the transaction. A Participant that has not responded with a Prepared vote can unilaterally abort the transaction. When the Coordinator has received votes from every Participant, or a pre-defined timeout has occurred, the Coordinator starts the second phase (the Commit/Abort phase) by notifying the Participants of the outcome of the transaction. The Coordinator decides to commit a transaction if it has received Prepared votes from all of the Participants during the first phase; otherwise, it decides to abort the transaction.

On the Coordinator-side, the services comprise:

- **Activation Service**: The Activation service creates a Coordinator object and a transaction context for each transaction. Essentially, the Activation service behaves like a factory object that creates Coordinator objects.
- **Registration Service**: The Registration service allows the Participants and the Initiator to register their endpoint references.
- **Completion Service**: The Completion service allows the Initiator to signal the start of the distributed commit.
- **Coordinator Service**: The Coordinator service runs the 2PC protocol, which ensures atomic commitment of the distributed transaction.

The Activation service is used for all transactions. It is provided by a single object, which is replicated in our BFT framework. When a distributed transaction is activated, a Coordinator object is created. The Coordinator object provides the Registration service, the Completion service and the Coordinator service. The transaction context contains a unique transaction id and an endpoint reference for the Registration service, and is included in all request messages sent during the transaction. The Coordinator object is replicated in our BFT framework.

At the Initiator, the service comprises:

- **CompletionInitiator Service**: The CompletionInitiator service is used by the Coordinator to inform the Initiator of the final outcome of the transaction, as part of the Completion protocol.

At a Participant, the service comprises:

- **Participant Service**: The Participant service is used by the Coordinator to solicit votes from, and to send the transaction outcome to, the Participant.

The phases of a distributed transaction using a conformant WS-AT framework are shown in Figure 1 with a banking application adapted from [4] and used in our performance evaluation (note that this example is for illustration purposes only and in practice, Accounts A and B might not be exposed as Web Services). In this example application, the bank provides an online banking Web Service that a customer can access. The transaction is started as a result of the customer's invoking a Web Service of the bank to transfer an amount of money from one account to another.

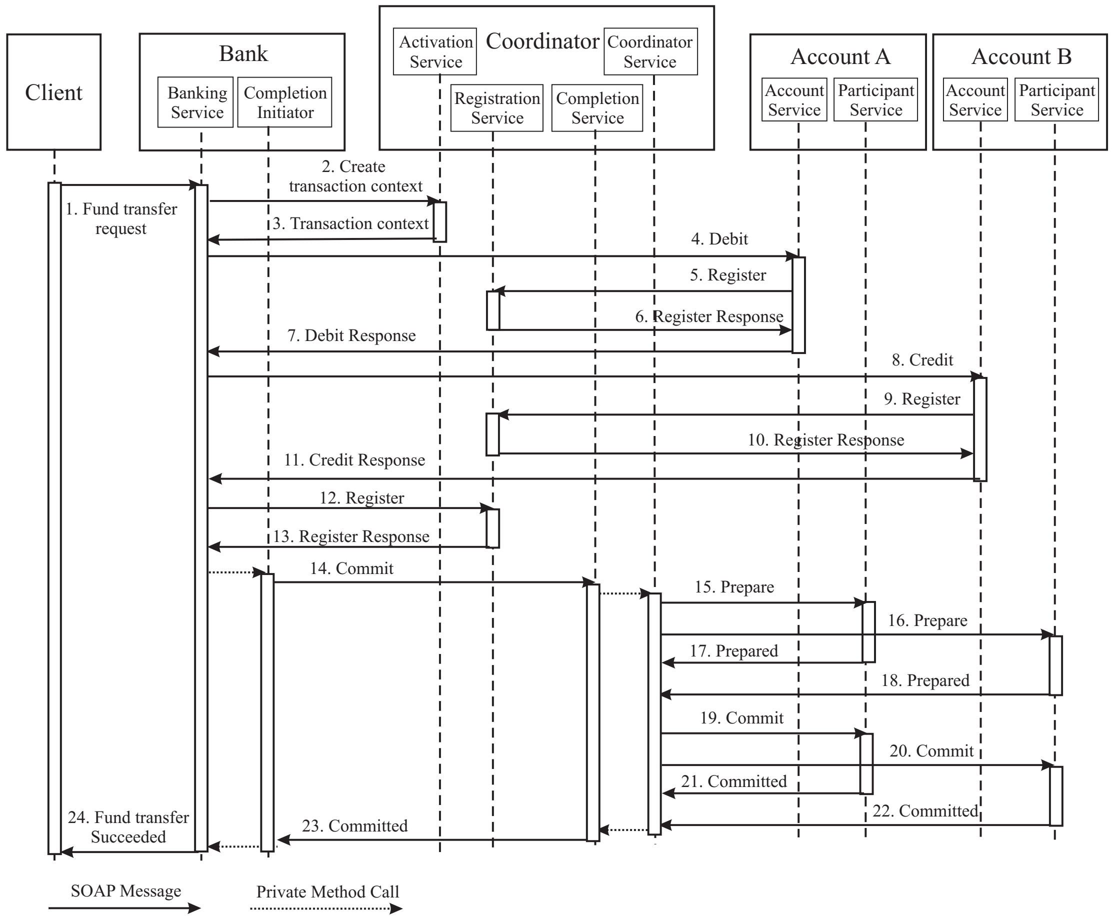


Fig. 1. The sequence diagram for the banking application using Web Services Atomic Transactions.

C. Byzantine Fault Tolerance

Byzantine fault tolerance (BFT) refers to the ability of a system to tolerate Byzantine faults. A *Byzantine fault* is an arbitrary fault, which might be a hardware fault, or a software fault caused by an intrusion into the system. Byzantine fault tolerance can be achieved by replicating the server and by ensuring that all server replicas reach agreement on each input despite Byzantine faulty replicas and clients. Such an agreement is referred to as a Byzantine Agreement (BA) [17].

In the implementation of our BFT framework, we use the Practical Byzantine Fault Tolerance (PBFT) algorithm of Castro and Liskov [9]. We provide here a synopsis of the PBFT algorithm during normal operation. The PBFT algorithm is executed by a set of $3f + 1$ replicas and tolerates f Byzantine faulty replicas. One of the replicas is the primary, and the rest of the replicas are the backups. The normal operation of the PBFT algorithm involves three phases. During the first phase (the Pre-Prepare phase), the primary multicasts, to the backups, a Pre-Prepare message containing the client's request, the current view number, and the sequence number of the request. A backup verifies the Pre-Prepare message and ordering information (used to order messages and non-deterministic operations) by checking the digital signature, the view number, and the sequence number. If the backup accepts the message, it starts the second phase (the Prepare phase) by multicasting, to the other replicas, a Prepare message containing the ordering information and the digest of the request message being ordered. A replica waits until it has collected matching Prepare messages from $2f$ other replicas. It then starts the third phase (the Commit phase) by multicasting a Commit message to the other replicas. The Commit phase ends when a replica has received matching Commit messages from $2f$ other replicas. At this point, the request message is ordered and ready to be delivered to the server application.

III. System Model

We consider a composite Web Service that utilizes individual Web Services provided by different organizations or departments within the same organization, similar to the example shown in Figure 1. We assume that an end user accesses the composite Web Service through a Web browser or invokes the Web Service directly through a stand-alone client application. For each request from an end user, a distributed transaction is created to coordinate the interactions with other Web Services. For simplicity, we assume a flat distributed transaction model (*i.e.,* a distributed transaction does not involve nested transactions [13]). The distributed transaction is supported by a conformant WS-AT framework such as the Kandula framework [4].

The composite Web Service provider is the Initiator of the transaction. We assume that the Initiator is stateless because typically it provides only a front-end service for the clients and delegates the work to the Participants. The Initiator starts and terminates the transaction. The Initiator also propagates the transaction to other Participants using a transaction context in the requests.

We assume that, for each transaction, a distinct Coordinator object is created. The lifetime of the Coordinator is the same as that of the transaction it coordinates. Moreover, we assume that the Coordinator runs separately from the Initiator and the Participants. Although it is common practice to collocate the Initiator with the Coordinator, such an approach might not be desirable for two reasons. First, collocating the Initiator and the Coordinator tightly couples the business logic with the transaction coordination mechanism, which is desirable neither from the software engineering perspective (it is harder to test) nor from the security perspective (it defies the defense-in-depth principle). Second, the Initiator typically is stateless and, thus, can be rendered fault tolerant more easily than the Coordinator, which is stateful. This difference naturally calls for the separation of the Initiator and the Coordinator.

We assume that the Initiator and the Coordinator are subject to Byzantine faults and, thus, they are replicated. To tolerate f faulty Coordinator replicas during a transaction, $3f+1$ Coordinator replicas are required. We assume that the Initiator is stateless and, hence, only $2f+1$ Initiator replicas are required to tolerate f faulty Initiator replicas.

The Participants in a transaction are not replicated and can be Byzantine faulty. In the presence of Byzantine faulty Participants, our BFT framework ensures that non-faulty Coordinator replicas agree on the same decision regarding the outcome of the transaction, as do non-faulty Participants. It is impossible to achieve atomic commitment for *all* Participants in a distributed transaction, if some of the Participants are Byzantine faulty. The 2PC protocol implicitly assumes that the Participants are trustworthy. That is, if a Participant votes to commit/abort a transaction, it is indeed prepared to do so, and it tries to commit a transaction whenever it can. A Byzantine faulty Participant might vote to abort a transaction, or to commit a transaction but abort the transaction locally, or it might choose not to register and not to participate in the two-phase commit. We do not attempt to address whether it makes sense to consider the possibility of a Byzantine faulty Participant. Furthermore, in most real-world applications, replication of the Participants is not very realistic. As for other BFT algorithms/protocols, we assume that the clients can be Byzantine faulty.

Each message exchanged between the Coordinator and the Initiator or a Participant is protected by a security token. The security token can be a digital signature, or a message authentication code.

We assume that each Coordinator replica and each Participant has a public/private key pair. The public keys of the Participants are known to all Coordinator replicas and vice versa, whereas their private keys are kept secret. We assume that each pair of entities that might exchange messages has a shared symmetric key, so that the entities can use a message authentication code as the security token. We assume that the adversaries have limited computing power, which prevents them from breaking the private encryption keys and the digital signatures of non-faulty replicas.

IV. BFT Coordination for WS-AT

Our BFT framework comprises a suite of protocols and mechanisms that protect the services and infrastructure of WS-AT coordination against Byzantine faults. The BFT Activation protocol ensures that the same transaction id for a transaction is chosen by non-faulty Activation replicas. The BFT Registration protocol involves no inter-replica communication and guarantees that a non-faulty Participant registers with at least $2f+1$ Coordinator replicas. The Byzantine agreement on the registration set (*i.e.*, the set of Participants involved in the transaction) is deferred to, and combined with, the agreement on the transaction outcome in the BFT Completion and Distributed Commit protocol.

There exist a number of well-known general-purpose BFT algorithms [1], [2], [7], [9], [16], [33]. In principle, most of them can be adapted to ensure Byzantine agreement in our BFT framework, although it is easier to adapt those that support the leader-follower-based replication approach, such as PBFT [9] or Zyvvya [16]. Instead of ensuring Byzantine agreement on the total ordering of requests, those algorithms can be easily modified to guarantee Byzantine agreement on the key values (*i.e.*, transaction id, registration set, and transaction outcome) needed for trustworthy WS-AT coordination. In our protocol descriptions, we assume such an adapted algorithm is used.

The liveness of our protocols also depends on the underlying (adapted) Byzantine agreement algorithm. Because the modification might not be obvious, we present here how we adapted the PBFT view change algorithm for our purposes.

Furthermore, for clarity in the descriptions and the associated diagrams of our protocols, we assume that the underlying Byzantine agreement algorithm requires $3f+1$ replicas to tolerate f faulty replicas, so that it is compatible with the requirement needed for other operations in our protocols. Our protocols will, of course, work if more than $3f+1$ replicas are used for Byzantine agreement.

In our protocols, a multicast message is in fact sent point-to-point from the sender to each intended receiver. As such, if digital signatures are not used, a message authentication code between the sender and each receiver is used, instead of an authenticator [9]. For brevity, we use the symbol σ to denote the token, which could be either a digital signature, or a message authentication code (without detailed indexes).

A. BFT Activation

The Activation service of WS-AT creates a new Coordinator object for each new transaction. Different Coordinator objects do not share data and their initial states are not dependent on the order in which they are created. Consequently, there is no need to order Activation request messages.

An important part of the state of the Coordinator object is the transaction id. The transaction id is unique, *i.e.*, no two transactions have the same transaction id. For security reasons, the transaction id is derived from a random number, so that the transaction id cannot be predicted and, thus, subverted, as described below. Byzantine agreement is needed to ensure that each non-faulty Activation replica assigns the same transaction id to a transaction.

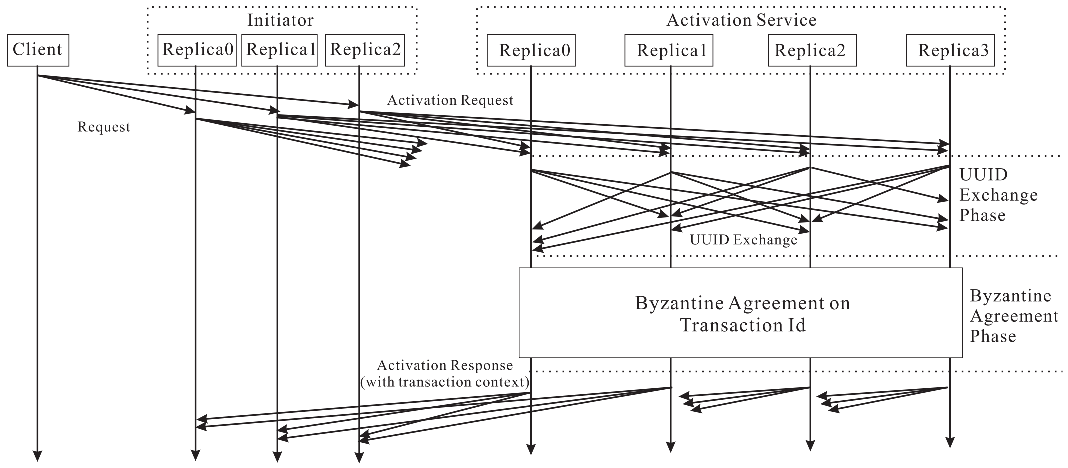


Fig. 2. BFT Activation protocol.

However, one should not trust the primary Activation replica to create the transaction id. Because the UUID is random, a backup Activation replica is in no position to verify that the UUID proposed by the primary is random, and cannot prevent a Byzantine faulty primary from using a deterministic algorithm to generate the UUID. Without additional security protection, the use of a predictable transaction id opens the door for an adversary to take over a non-faulty Coordinator and hijack a communication session with the Participants, similar to the Mitnick attack [25] (the transaction id is analogous to the initial TCP sequence number). Such hijacking can lead to non-atomic transaction commitment at different Participants (and other undesirable consequences). Even though this type of attack can be defeated by applying other security mechanisms, such as digital signatures, the use of a random transaction id during activation is a good defense-in-depth strategy.

To cope with the use of random numbers for Activation, our BFT protocol employs an additional round of message exchange to ensure that the final random number used to generate the transaction id is collectively determined by a quorum of Coordinator replicas. The collective determination of the transaction id (instead of delegating the task to the primary replica) is important because it prevents a Byzantine faulty primary from colluding with an adversary to cause non-atomic transaction commitment.

Figure 2 shows our BFT protocol for the Activation service. A non-faulty client sends a request message to all Initiator replicas. The request message 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's id, and σ_c is c's security token for the message.

An Initiator replica accepts the request and sends back a reply (when the transaction is completed) if the request is properly protected by a security token, and the Initiator replica has not already accepted a request with a timestamp that is greater than or equal to t from the same client. If the request carries an obsolete timestamp, the Initiator replica retransmits the reply if it finds the reply in its message log. The client does not deliver the reply until it has collected $f+1$ matching replies from the Initiator replicas.

On receiving a request message from a client, an Initiator replica starts a distributed transaction and multicasts an Activation request to all Activation replicas. The Activation request has the form $<\text{ACTIVATION}, v, t, c, k>_{\sigma_k}$, where v is the current view number, t is the timestamp, c is the client's id, k is the Initiator replica's id, and σ_k is k's security token for the message.

An Activation replica logs the Activation request if the message is properly protected by a security token and the Activation replica has not accepted a request with a timestamp that is greater than or equal to t from the Initiator replica in view v. When the Activation replica receives Activation request messages, with matching t and c, from $f+1$ distinct Initiator replicas, it accepts the Activation request. This requirement ensures that the Activation request comes from at least one non-faulty Initiator replica.

An Activation replica then generates a UUID and multicasts a UUID Exchange message to all of the other replicas. The message has the form $<\text{UUID-EXCHANGE}, v, id, d, uuid_i, i>_{\sigma_i}$, where v is the current view, id is the tuple $<t, c>$ that identifies the activation instance, d is the digest of the Activation request, $uuid_i$ is replica i's proposed UUID, i is the replica's id, and σ_i is i's security token for the message.

An Activation replica accepts a UUID Exchange message if the replica is in view v, the message is properly protected by a security token, and d matches the digest of the Activation request. When the Activation primary replica has generated its own UUID Exchange message and has accepted UUID Exchange messages from $2f$ backup replicas, it combines the set of $2f+1$ UUIDs by performing a bit-wise exclusive-or operation on the set of UUIDs, which is a provably secure combination method [35]. It then initiates a Byzantine agreement for the set of $< i, uuid_i >$ tuples, together with the combined UUID.

At the end of the Byzantine agreement, a replica derives the transaction id tid from the combined UUID, creates a Coordinator object for the tid, and sends an Activation response to the Initiator replicas. The Activation response has the form $<\text{ACTIVATION-REPLY}, t, c, v, C, i>_{\sigma_i}$, where t and c are the timestamp and client id included in the Activation

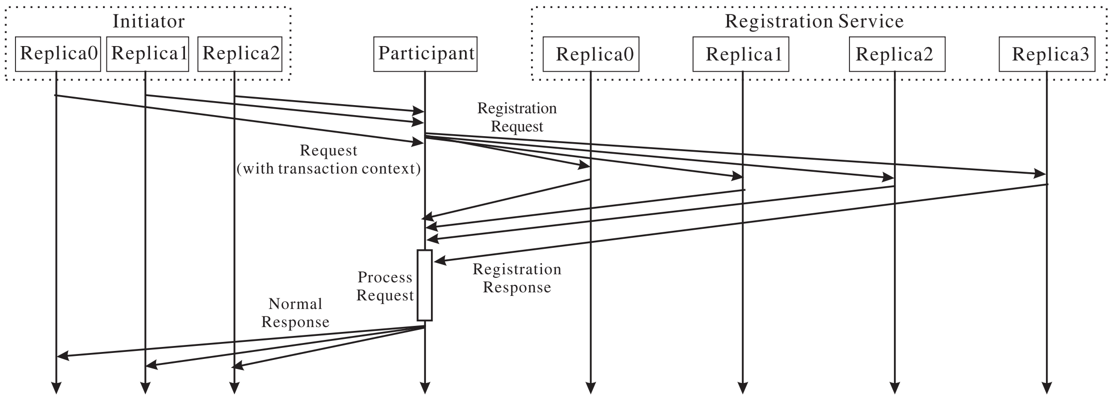


Fig. 3. BFT Registration and Transaction Propagation protocol.

request, v is the view number, C is the transaction context, i is the replica's id, and σ_i is i's security token for the message.

An Initiator replica logs the Activation response if it is properly signed, and t and c match those in the Activation request. The Initiator replica accepts the Activation response if it has logged matching Activation responses from $f + 1$ distinct Activation replicas. Doing so ensures the validity of the Activation response because it comes from at least one non-faulty replica.

B. BFT Registration and Transaction Propagation

To ensure atomic termination of a distributed transaction, it is essential that all non-faulty Coordinator replicas agree on the set of Participants involved in the transaction. Such agreement can be achieved by running a Byzantine agreement instance among the Coordinator replicas whenever a Participant registers itself. However, doing so incurs too much overhead to be practical. Consequently, we defer the Byzantine agreement on the Participant set until the Distributed Commit phase and combine it with the Byzantine agreement for the transaction outcome. During the Registration phase, we run a lightweight BFT Registration protocol with no inter-replica communication, as shown in Figure 3.

In our BFT Registration protocol, a Participant accepts a request from an Initiator replica when it has collected matching requests from $f + 1$ distinct Initiator replicas. One of those messages must have been sent by a non-faulty Initiator replica, because at most f Initiator replicas are faulty. Consequently, a faulty Initiator replica is prevented from excluding a Participant from the transaction (*e.g.*, by not including it in the transaction context in the request), or from including a process that should not be a Participant.

To register, a Participant multicasts a Registration request to the Coordinator replicas and waits to receive acknowledgments from $2f + 1$ distinct Coordinator replicas. Because at most f Coordinator replicas are faulty, at least $f+1$ non-faulty Coordinator replicas must have accepted the Registration request.

If a Participant registers successfully and completes its execution of the Initiator's request, it multicasts a normal response to the Initiator replicas. Otherwise, it multicasts an exception (possibly after recovering from a transient fault) to the Initiator replicas. If an Initiator replica receives an exception from a Participant, or times out a Participant, it aborts the transaction.

An Initiator replica also registers with the Registration service. This registration is similar to that of the Participants. Because at most f Initiator replicas are faulty, at least $f + 1$ Initiator replicas register successfully.

C. BFT Completion and Distributed Commit

The normal operation of the BFT Completion and Distributed Commit protocol is illustrated in Figure 4. When an Initiator replica successfully completes all operations within a transaction before a timeout, it multicasts a Commit request to the Coordinator replicas; otherwise, it multicasts a Rollback request. A Coordinator replica accepts the Commit or Rollback request when it has received matching requests from $f + 1$ distinct Initiator replicas.

On accepting a Commit request, a Coordinator replica starts the Registration Update phase by multicasting, to the Coordinator replicas, a Registration Update message including the registration records of the Participants known to itself. The purpose of this phase is to ensure that, if a non-faulty Participant has registered successfully, all non-faulty Coordinator replicas have a record of its registration. The Registration Update message has the form $<\text{REGISTRATION-UPDATE}, tid, RS, i>_{\sigma_i}$, where tid is the transaction id, RS is the set of registration records, i is the replica's id, and σ_i is i's security token for the message. Each record in the set RS is a registration record $R_j = (tid, j)_{\sigma_j}$, where j is the Participant id.

When a Coordinator replica has collected Registration Update messages from $2f$ other replicas, it checks to see if it has missed any registration records. (It is possible that a replica has missed a registration record, because a Participant is required to obtain acknowledgments from only $2f + 1$ Coordinator replicas.) The Coordinator replica adds a missing registration record if the record is present in the RS set. The Registration Update phase is then concluded.

Next, the Coordinator replica starts the first (Prepare) phase of the standard 2PC protocol. A Participant defers accepting a Commit/Rollback request until it has collected matching requests from $f + 1$ distinct Coordinator replicas to ensure that the request comes from a non-faulty Coordinator replica. At the end of the first phase of the 2PC protocol, the Byzantine

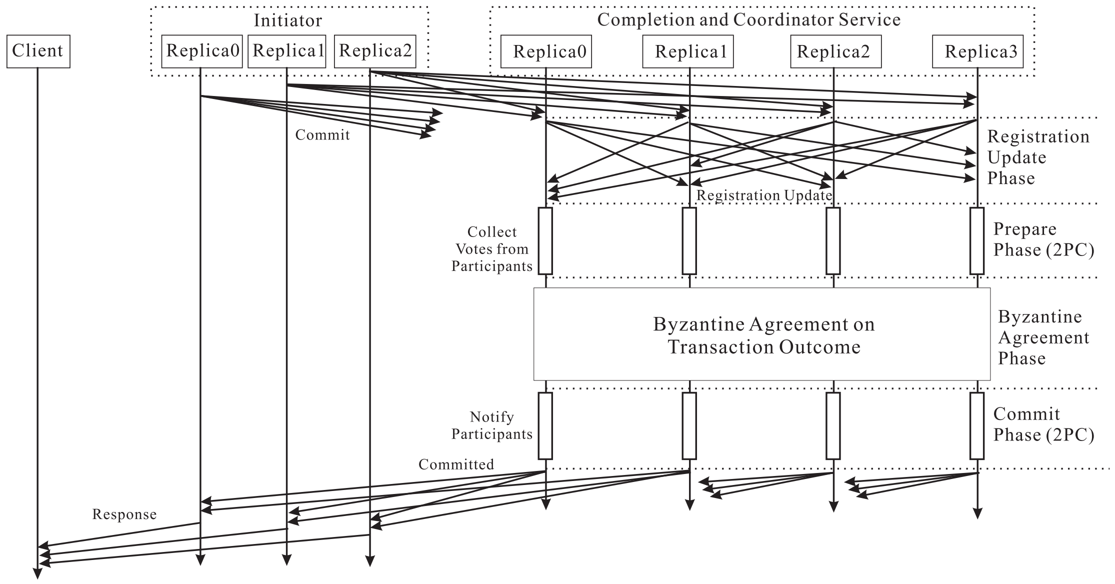


Fig. 4. BFT Completion and Distributed Commit protocol.

agreement algorithm is run, so that all non-faulty Coordinator replicas agree on the same set of Participants and the same transaction outcome. This phase is followed by the second (Commit/Abort) phase of the 2PC protocol. If the Coordinator replica receives a Rollback (*i.e.,* an Abort) request, the first phase of the 2PC protocol is skipped because its result is Abort. However, the Byzantine agreement phase is still needed to ensure that all non-faulty replicas agree on the same transaction outcome.

When the Distributed Commit phase is completed, the Coordinator replicas notify the Initiator replicas of the transaction outcome. An Initiator replica accepts a Commit/Abort notification message if it has collected matching Commit/Abort notification messages from $f+1$ distinct Coordinator replicas. Similarly, a Participant accepts a Commit/Abort notification message if it has collected matching Commit/Abort notification messages from $f+1$ distinct Coordinator replicas. Doing so ensures that the Commit/Abort notification message comes from a non-faulty Coordinator replica.

For Byzantine agreement, the primary Coordinator replica must include its decision certificate C as evidence for its decision on the transaction outcome, where C contains a set of records, one for each Participant. The record for Participant j contains a registration record $R_j = (tid, j)_{\sigma_j}$ and a vote record $V_j = (tid, vote)_{\sigma_j}$. The tid is included in each registration record and vote record, so that a faulty primary Coordinator replica cannot re-use an obsolete registration or vote to force a transaction outcome against the will of a non-faulty Participant.

A backup Coordinator replica suspects the primary Coordinator replica and initiates a view change unless the registration records in C are either identical to, or form a superset of, its local registration records, and the proposed transaction outcome is consistent with the registration and vote records.

At the end of the Byzantine agreement phase, a Coordinator replica runs the second (Commit/Abort) phase of the 2PC protocol, where the decision outcome is sent to each Participant. A Participant accepts a decision notification when it has collected matching decision notifications from $f+1$ distinct Coordinator replicas.

D. View Changes

The Activation service and the Completion and Coordinator services each employ a primary replica and use inter-replica communication. For these services, a View Change algorithm is necessary to maintain liveness. Neither the Initiator object nor the Registration service have a primary replica, and they do not use inter-replica communication. Thus, a View Change algorithm is not necessary for the Initiator object and the Registration service.

To prevent a faulty primary from hindering the liveness of the Activation protocol or the Completion and Distributed Commit protocol, or disseminating conflicting information to different replicas, a View Change algorithm is used. A backup replica initiates a view change when it cannot advance to the next phase within a reasonable time, or when it detects that the primary has sent conflicting information. A View Change algorithm is used to select a new primary when the existing primary is suspected to be Byzantine faulty.

Because the modifications needed to adapt an existing View Change algorithm might not be obvious, we elaborate how to adapt the View Change algorithm of PBFT [8]. Like [16], we choose to use digital signatures rather than message authentication codes to protect messages exchanged for the view change because, with message authentication codes, the View Change algorithm is much more complicated and achieves only a little less latency. The additional overhead of using digital signatures for View Change messages is relatively small because, typically, the fault detection time (*i.e.,* view change timeout value) is much larger than the latency for a successful view change round. If the first attempt at view change does not complete, then the fault detection time needs to be increased.

According to PBFT [8], on suspecting the primary, a replica multicasts a View Change message. On receiving a View Change message, a replica that has not suspected the primary logs the message if it determines that the message is valid. When the replica has received valid View Change messages from $f+1$ distinct replicas, it multicasts its own View Change message even if it has not suspected the primary. When the primary for the new view receives valid View Change messages for the new view from $2f + 1$ distinct replicas (including itself), in installs the new view and multicasts a New View message.

1) BFT Activation: For the BFT Activation protocol, the View Change message has the form $<\textsc{view-change}, v + 1, tid, P, i>_{\sigma_i}$, where $v+1$ is the new view number, tid is the transaction id, P contains information regarding the current state of the replica, i is the replica's id, and σ_i is i's signature for the message. The contents of P are defined in terms of $id =< t, c >$ (the identifier of the activation instance, with timestamp t and client id c) and $uuid_i$ (the UUID proposed by replica i), as follows:

– If replica i has not yet reached the BA-Pre-Prepared state in view $v' \leq v$, P is the tuple $<v', id, uuid_i>$.
– If replica i has reached the BA-Pre-Prepared state in view $v' \leq v$, P is the tuple $<v', id, U>$, where U is the set of tuples $<uuid_i, i>$ originally sent by $2f + 1$ replicas and included by the primary of view v' in the BA-Pre-Prepare-Update message.
– If replica i has reached the BA-Prepared state in view $v' \leq v$, P contains the tuple $<v', id, U>$ and the matching BA-Prepare messages sent by $2f$ other replicas in view v'.

The primary of view $v+1$ accepts a View Change message, provided that $v' \leq v$, tid is the expected transaction id, and the message is properly signed. If the primary of view $v+1$ finds that a BA-Pre-Prepare-Reply message is missing, it requests retransmission from the corresponding replica.

When the primary of view $v + 1$ has collected View Change messages from $2f$ other replicas, it reconstructs the state (described below), installs the new view $v + 1$, and multicasts a New View message. The New View message has the form $<\textsc{new-view}, v + 1, tid, V, U, p>_{\sigma_p}$, where $v + 1$ is the new view number, tid is the transaction id, V contains the tuples for the View Change messages received from $2f + 1$ distinct replicas for view $v + 1$, U reflects the constructed state (described below), p is the primary's id, and σ_p is p's signature for the message. Each tuple in V has the form $<i, d>$, where i is the replica's id and d is the digest of the corresponding View Change message. The contents of U are determined as follows:

– If the new primary has received a valid View Change message containing a BA-Prepare record, it uses the same U as in the BA-Prepare record, which is the set of $<uuid_i, i>$ tuples from $2f + 1$ distinct replicas.
– If the new primary has received one or more valid View Change messages containing a BA-Pre-Prepare record, it selects the one with the highest view number and uses the U in that View Change message.
– Otherwise, the new primary constructs U as the set of $<uuid_i, i>$ tuples based on the View Change messages it received from $2f + 1$ distinct replicas.

When a backup receives a New View message, it verifies the message by following essentially the same steps used by the primary to determine U. If the backup accepts the New View message, it multicasts a BA-Prepare message to the other replicas. It then proceeds as in normal operation.

2) BFT Completion and Distributed Commit: For the BFT Completion and Distributed Commit protocol, the View Change message has the form $<\textsc{view-change}, v + 1, tid, P, i>_{\sigma_i}$, where $v + 1$ is the new view number, tid is the transaction id, P contains information (described below) regarding the current state of the replica, i is the replica's id, and σ_i is i's signature for the message. The contents of P are determined as follows:

– If replica i has not reached the BA-Pre-Prepared state in view $v' \leq v$, it uses its own decision certificate C as P.
– If replica i has reached the BA-Pre-Prepared state in view $v' \leq v$, P is the tuple $<v', tid, O, C>$, where v' is the view number, tid is the transaction id, O is the transaction outcome, and C is the decision certificate proposed by the primary in view v'.
– If replica i has reached the BA-Prepared state in view $v' \leq v$, P contains the tuple $<v', tid, O, C>$ and the matching BA-Prepare messages from $2f$ distinct replicas in view v'.

The primary of view $v+1$ accepts a View Change message if $v' \leq v$, tid is the expected transaction id, and the message is properly signed.

When the primary of view $v+1$ has collected View Change messages from $2f$ other replicas, it reconstructs the state (described below), installs the new view $v+1$, and multicasts a New View message. The New View message has the form $<\textsc{new-view}, v + 1, tid, V, O, C>_{\sigma_p}$, where $v + 1$ is the new view number, tid is the transaction id, V contains the tuples for the View Change messages received from $2f + 1$ distinct replicas for view $v + 1$, O and C reflect the constructed state (described below), and σ_p is p's signature. Each tuple in V has the form $<i, d>$, where i is the sending replica's id and d is the digest of the corresponding View Change message. The contents of O and C are determined as follows:

– If the new primary has received one or more valid View Change message containing a BA-Prepare record, and none of those BA-Prepare records conflicts, it uses the same outcome O and the same decision certificate C in the New View message as in the BA-Prepare record.
– Otherwise, the new primary constructs a set of voting records by examining the voting records in the decision certificate in the View Change message. If the new primary finds a voting record that it did not receive, it adds the record to the decision certificate C. If the new primary finds conflicting voting records from the same Participant (*i.e.*, a Participant sent a Prepared vote to one Coordinator replica and an Abort vote to another Coordinator replica), it aborts the transaction (*i.e.*, it sets the outcome O to Abort and includes the conflicting votes from the offending Participant in the decision certificate C as evidence of foul play), because the integrity of the

transaction can no longer be guaranteed.

When a backup receives the New View message, it verifies the message by following basically the same steps used by the primary. If the backup accepts the New View message, it multicasts a BA-Prepare message to the other replicas. It then proceeds as in normal operation.

V. Performance Evaluation

We have implemented our BFT protocols and mechanisms for Web Services Atomic Transactions (WS-AT) and incorporated them into the Kandula framework [4], which is a Java-based, open-source implementation of the WS-Transaction 1.1 specification, including both WS-AT and Web Services Business Activities (WS-BA). The extended framework uses the WSS4J implementation (version 1.6) of the Web Services Security Specification [5] and the Apache Axis 1.1 SOAP Engine [3]. Most of the BFT mechanisms are implemented using Axis handlers that are plugged into the framework without affecting other components. Some of the Kandula code is modified to enable control of its internal state and BFT delivery of requests at the Initiator, the Coordinator and the Participants.

For Byzantine fault tolerance, all messages exchanged are protected with a message authentication code (referred to as the HMAC digital signature in WSS4J) during normal operation with the exception of the vote messages from the Participants. The vote messages are signed to ensure accountability, with timestamped digital signatures (referred to as the RSA digital signature in WSS4J) during view changes. For the HMAC digital signature, SHA1 is used. For the RSA digital signature, the key size is 1024 bits (SHA1 is used for the message digest).

To demonstrate the benefits of using our BFT framework for WS-AT over the naive application of a traditional BFT algorithm, we implemented an adapted version of the Practical Byzantine Fault Tolerance (PBFT) algorithm [9]. In this reference BFT implementation, in addition to a round of Byzantine agreement for the Activation and Registration requests, a round of Byzantine agreement is used for each vote message from a Participant during the first phase of the two-phase commit protocol, to ensure consistency among the Coordinator replicas.

We have evaluated the performance of our BFT framework both in a Local-Area Nework (LAN) testbed and in a Wide-Area Network (WAN) testbed (*i.e.,* PlanetLab). The LAN testbed consists of 14 HP BL460c blade servers connected by a Cisco 3020 Gigabit switch. Each blade server is equipped with two Xeon E5405 (2 GHz) processors and 5 GB RAM, and runs the 64-bit Ubuntu Linux server operating system. For the fault scalability experiment, up to four virtual machines are launched on each physical node, because we do not have an adequate number of physical nodes for experiments with higher replication degree. Each virtual machine runs the 32-bit Ubuntu Linux server operating system with 1GB RAM allocated to it.

The hardware specifications of the PlanetLab nodes vary significantly. The nodes we chose to use are generally equipped with Intel Core 2 Duo CPUs (2GHz to 2.6GHz). Note that the nodes in PlanetLab are shared among many users, and we have no control over the actual load on the CPU and available physical memory.

The test application is the banking application, described in Section II-B. There are three Initiator replicas, hosted on three distinct nodes, and four Coordinator replicas, hosted on four distinct 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 consecutive calls. The clients’ request messages and the corresponding response messages range from 3KB to 10KB in size. In each run, 1,000 samples are obtained. The end-to-end latency for the fund transfer operation is measured at a client. The latencies for the Activation and Distributed Commit services are measured at the primary Coordinator replica. The throughput of the Distributed Commit service is measured at an Initiator replica for various numbers of Participants and concurrent clients.

A. Performance Evaluation in a LAN

End-to-end latency and throughput for $f = 1$. The end-to-end latency results in the LAN for $f = 1$ are shown in Figure 5(a). These results are obtained for four different configurations:

(1) The test application is used as is, without any modification (labeled “Unmodified Application” in the figure).
(2) The test application is modified so that all messages exchanged within each transaction are protected by a message authentication code (labeled “With HMAC” in the figure).
(3) The test application with the Coordination services is protected by our BFT mechanisms (labeled “With Our BFT” in the figure).
(4) The test application with the Coordination services is protected by naively applying the adapted PBFT algorithm in the reference implementation (labeled “With Ref. BFT” in the figure).

In all four configurations, the end-to-end latency is measured in the presence of a single atomic transaction.

As can be seen from Figure 5(a), the end-to-end latency with our lightweight BFT framework is significantly greater than that for the unmodified application. However, this increase is mainly due to the use of the message authentication code, as revealed by the high end-to-end latency when only the HMAC signature is used. Thus, we use this configuration as the baseline for comparison. The high cost of the HMAC signature is somewhat surprising. We believe that this high cost is due to the use of WSS4J. In WSS4J version 1.6, the same APIs are provided to sign and verify an XML document using either the HMAC signature or the RSA signature. Unnecessary operations must have been introduced for the HMAC signature by that design choice.

As expected, the end-to-end latency for the reference BFT implementation is much greater than that for our BFT framework, as shown in Figure 5(a). For each transaction, the reference BFT implementation incurs the following *additional*

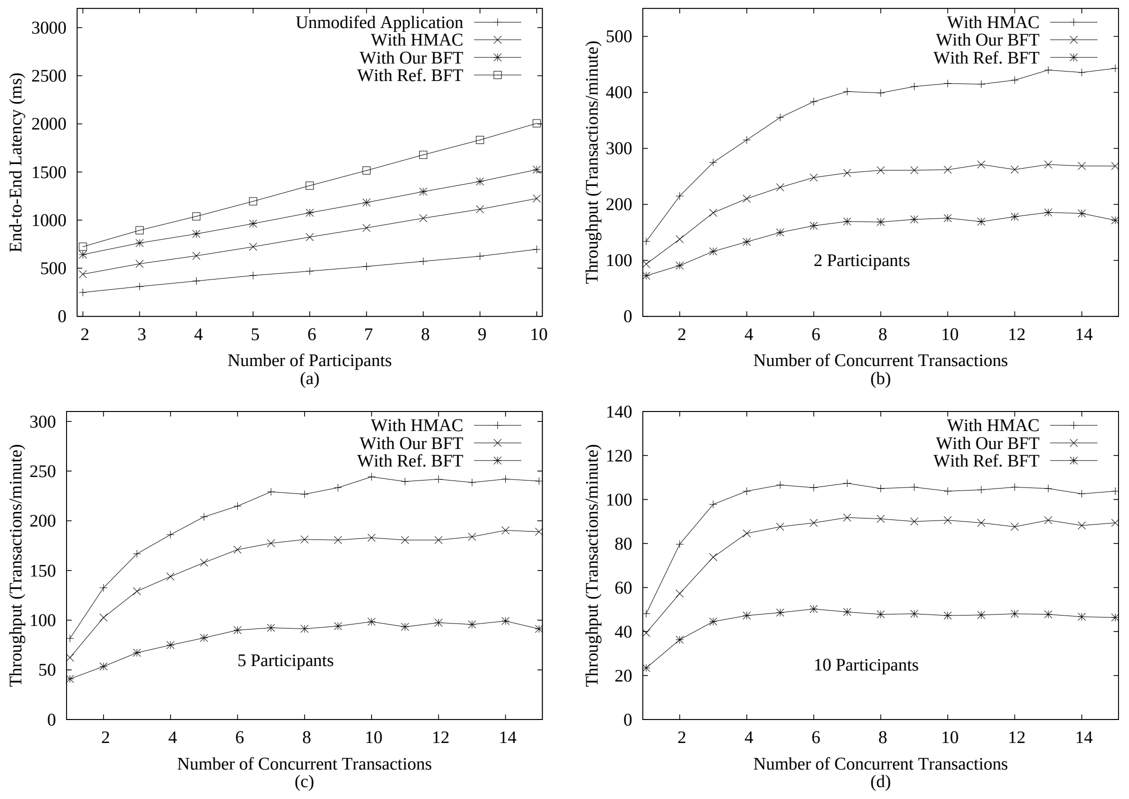


Fig. 5. End-to-end latency and throughput of the test application in a LAN.

overhead: (1) one round of Byzantine agreement for each Registration request, which our BFT framework avoids, (2) one round of Byzantine agreement for each vote message during two-phase commit, compared to our BFT framework which requires only one Byzantine agreement for each transaction regardless of the number of Participants. Consequently, as the number of Participants increases, the absolute overhead of the reference BFT implementation increases much more than that of our BFT framework. The relative overhead of our BFT framework (with respect to the performance of configuration (2)) ranges from about 45% with two Participants per transaction to about 25% with ten Participants per transaction. On the other hand, the relative overhead for the reference BFT implementation hovers around 100% in all cases.

The throughput results in the LAN for $f = 1$, in number of transactions per minute, for transactions with two, five and ten Participants are shown in Figure 5(b)-(d). Compared with the baseline configuration (the application with HMAC), the throughput reduction for our BFT framework ranges from about 45% with two Participants per transaction to about 15% with ten Participants per transaction. This reduction is expected because of the increased processing time at the Coordinator (for message protection and verification). Not surprisingly, the throughput reduction with the reference BFT implementation ranges between 50% and 60%, which is much more significant than the throughput reduction for our BFT framework.

Fault scalability. The fault scalability of our BFT framework is assessed with f (*i.e.*, the number of faults tolerated) between 1 and 5 (*i.e.*, the total number of Coordinator replicas is 4, 7, 10, 13 to 16). The end-to-end latency scalability result is obtained using a single client that issues one transaction at a time. As can be seen from Figure 6(a), the end-to-end latency increases slightly non-linearly with f. The increase in the latency with f arises from two factors: (1) the increased number of security operations (message digital signing and verification), and (2) the increased time to form a quorum. Note that the use of virtual machines has a significant negative impact on the overall performance. Compared to the $f = 1$ results obtained using the physical nodes, the end-to-end latency increased by approximately 35% (for $f = 1$), and the peak throughput (for $f = 1$) is reduced by approximately 20% (as shown in Figure 6(b). We suspect that this non-linear increase in the latency for large values of f is probably due to resource contention (the number of processing cores and available physical memory are much more limited in a virtual machine). To avoid clutter, we show only the results for two, five and ten Participants (per transaction) in Figure 6.

The peak throughput fault scalability results, shown in Figure 6(b), are obtained by varying the number of concurrent transactions. The sharp reduction in throughput as f increases for the two Participants and five Participants per transaction cases is somewhat unexpected. Further profiling of our prototype reveals that the time it takes to digitally sign or verify a message in the virtual machine is about 2-3 times that in the physical node. Consequently, the throughput is severely limited by the higher number of security operations for larger values of f. One might wonder why the throughput reduction is much less prominent for the ten Participants per transaction configuration. This is because the runtime overhead incurred

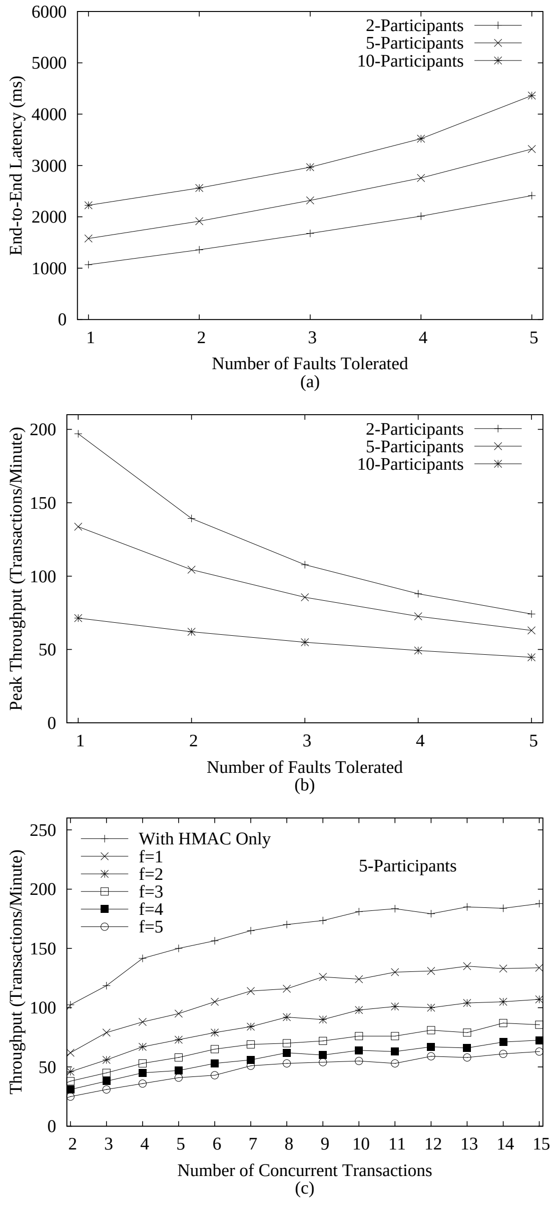


Fig. 6. Fault scalability results in a LAN.

at a replica for the same f is similar regardless of the number of Participants (*i.e.*, the overhead is only weakly dependent on the number of Participants), and the throughput for the ten Participants per transaction configuration is already quite low.

The throughput results with respect to different numbers of concurrent transactions for $f = 1$ to 5 (only the five Participants per transaction case is shown to avoid clutter) is shown in Figure 6(c). Because the performance using the virtual machines is quite different from that using the physical nodes, the baseline throughput results (*i.e.*, non-replicated with HMAC only) are also shown for comparison.

Fault recovery. The fault recovery time for the primary replica includes both the time to detect failure of the primary and the time for the primary view change. The fault recovery time depends heavily on the timeout value used to detect failure of the primary. In the LAN experiments, we used a timeout value of 500ms. The observed fault recovery time ranges from about 550ms to about 1sec (from when the primary process is killed until a new view is installed). This large jitter is mainly due to the uncertainty in the detection time of the primary failure by the different replicas. To eliminate this uncertainty and highlight the operational cost of our View Change protocol, we introduced an additional control interface so that a replica can be triggered to initiate a view change remotely.

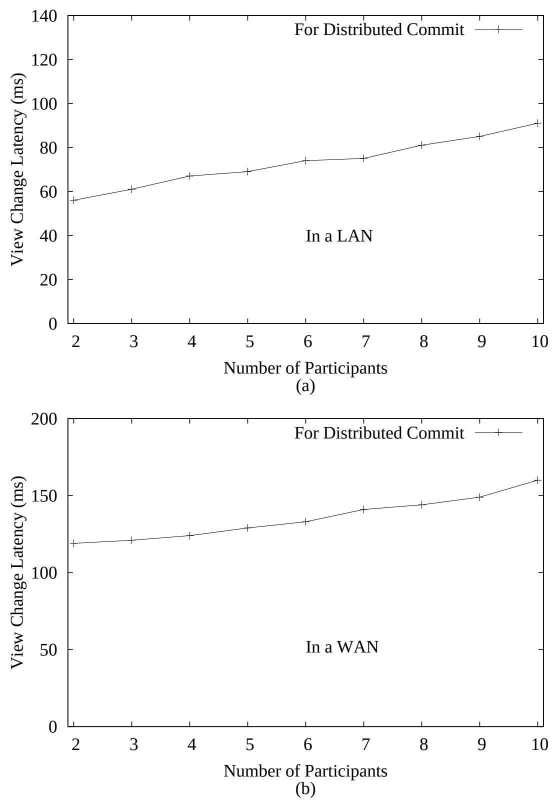


Fig. 7. View change latency (a) in a LAN and (b) in a WAN.

In the experiments, a controlling process kills the primary process and immediately directs all remaining replicas to initiate a view change. The view change latency is measured at a backup replica (for the new view) by finding the difference between the time the replica sent a View Change message and the time it received and verified the New View message. The experiments are performed for two different scenarios: (1) the primary is killed during Activation, and (2) the primary is killed during Distributed Commit. The results for the view change latency with different numbers of Participants are summarized in Figure 7(a). For scenario (1), the latency is independent of the number of Participants and is about 52ms. For scenario (2), the latency increases slightly with the number of Participants, because both the View Change message and the New View message are larger when there are more Participants. Note that the view change latency does not include the fault detection time.

B. *Performance Evaluation in a WAN*

In the WAN experiments, four PlanetLab nodes located at four distinct locations (harvard.edu, howard.edu, mcgillplanetlab.org, colostate.edu) are chosen to run the Coordinator

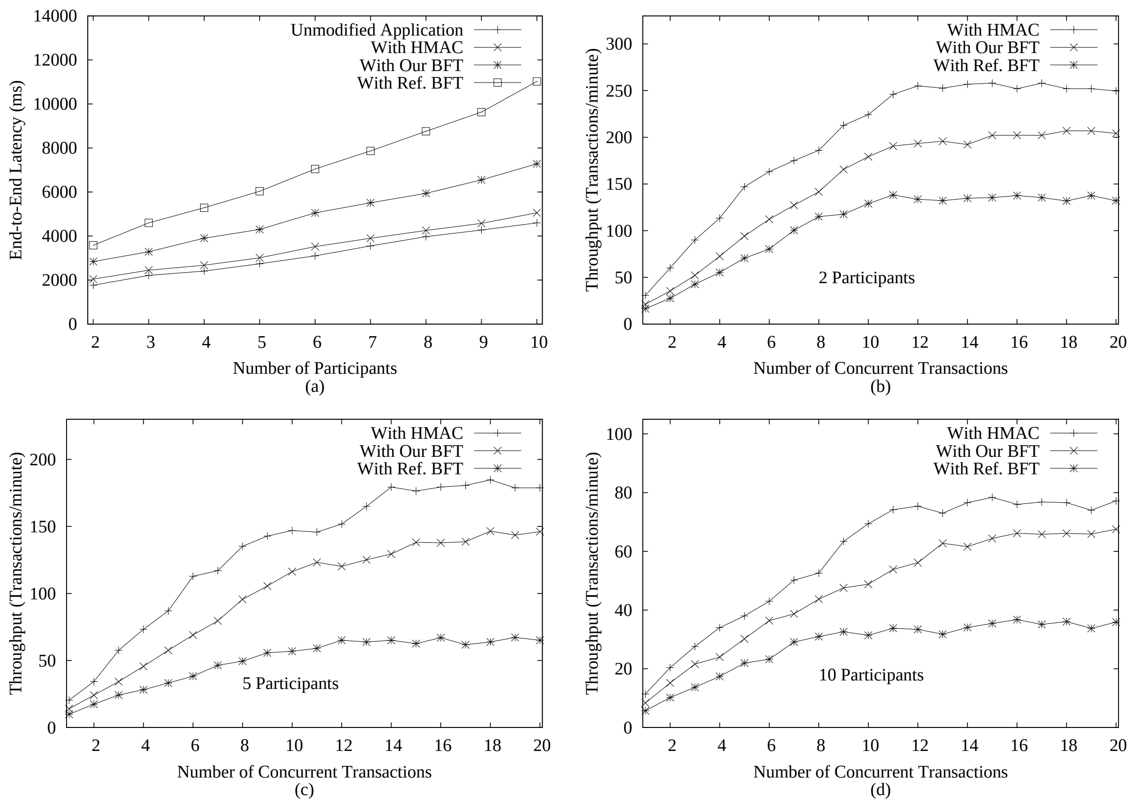


Fig. 8. End-to-end latency and throughput of the test application in a WAN.

replicas. The average round-trip time (as measured by the "ping" program) between the primary and each backup is approximately $40ms$. Similarly, PlanetLab nodes located at various locations are chosen to run the Participants. The round-trip time between a replica and a Participant varies from $20ms$ to $100ms$.

End-to-end latency and throughput for $f = 1$. The end-to-end latency results in the WAN for $f = 1$ are summarized in Figure 8 (a). As can been seen in the figure, the relative overhead for our BFT framework compared to the baseline is similar to that in the LAN (about 40%), and the same is true for the reference BFT implementation.

The throughput results in the WAN for $f = 1$ are shown in Figure 8(b)-(d). The baseline throughput is reduced by about 25% to 45% compared with that in the LAN due to the greater message transmission time (the network bandwidth in PlanetLab is significantly less than that in our LAN testbed) and the longer processing time (the CPUs in PlanetLab are generally less powerful than those in our LAN testbed). Thus, the throughput reduction with respect to the baseline is more moderate for both our BFT framework and the reference BFT implementation. For our BFT framework, the throughput reduction ranges from 15% to 20%. For the reference BFT implementation, the throughput reduction ranges from 35% to 55%.

Fault scalability. The fault scalability results are summarized in Figure 9. Similar to the LAN results, the end-to-end latency increases non-linearly with f (shown in Figure 9(a)); however, the increase in the latency with f is much more prominent compared with that in the LAN. We attribute the larger increase in the latency with f to the much increased time to form a quorum for larger values of f, because the communication delay in the WAN is several orders of magnitude larger than that in the LAN. This speculation is supported by the fact that much larger latency is observed when there are more Participants.

The peak throughput is somewhat less dependent on f compared with the end-to-end results, as shown in Figure 9(b). This is because unlike the end-to-end latency, the peak throughput is not sensitive to the long communication delay in the WAN. For completeness, the throughput results with respect to different numbers of concurrent transactions for $f = 1$ to 5 for five Participants are shown in Figure 9(c).

Fault recovery. As in the LAN, the fault recovery time for the primary replica in the WAN, without using the special control interface (see Section V-A), is largely determined by the timeout value used to detect failure of the primary. Due to the greater uncertainty of communication, we chose to use 2sec as the timeout value. This choice leads to a large fault recovery time, ranging from about 2sec to several seconds. With the special control interface, the view change latency (which does not include the fault detection time) ranges from about 110ms (for Activation) to about 160ms (for Distributed Commit) with ten Participants per transaction, as shown in Figure 7(b).

VI. Related Work

The problem of BFT distributed commit for atomic transactions has been previously addressed [22], [29]. Mohan *et al.* [22] proposed the first such protocol, which extended the

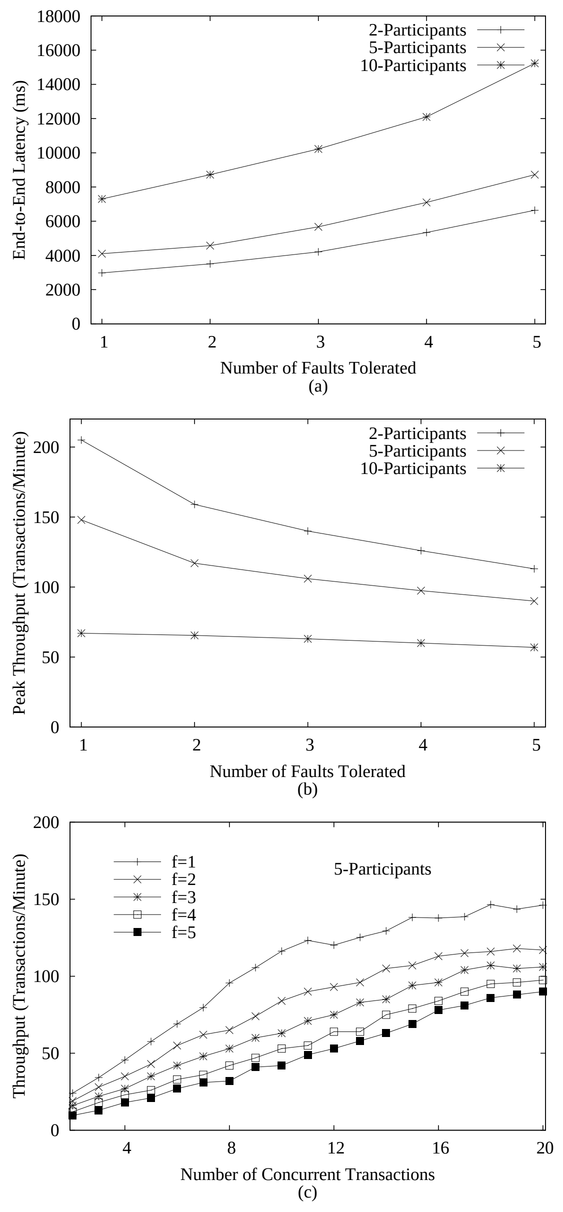


Fig. 9. Fault scalability results in a WAN.

2PC protocol with a Byzantine agreement phase on the transaction outcome among the Coordinator and the Participants in a root cluster. Such an approach has several limitations. First, transaction atomicity is guaranteed only for Participants in the root cluster. Second, every Participant in the root cluster must know the cluster membership, which might not be the case for Web Services Atomic Transactions, because one Participant does not necessarily know the other Participants. In contrast, our BFT framework requires Byzantine agreement among only the Coordinator replicas. Rothermel *et al.* [29] addressed the challenges of ensuring distributed commit for atomic transactions in open distributed systems, where Participants can be compromised. They assume that the root Coordinator is trusted, which obviates the need to replicate the Coordinator for Byzantine fault tolerance. However, such an assumption does not apply to Web Services that operate over the untrusted environment of the Internet.

There exists a number of general-purpose Byzantine fault tolerance algorithms/protocols [1], [2], [7], [8], [9], [10], [16], [23], [33], and group communication systems that handle Byzantine faults [12], [15], [24], [28]. Although they can be readily used to ensure BFT transaction coordination for WS-AT, they would incur unacceptable overhead if they were used naively, as shown in this paper. Essentially, our work demonstrates how to minimize the number of Byzantine agreement instances needed in a transaction by exploiting the semantics of Web Services Atomic Transactions and, thus, significantly improves the performance.

The system-level research closest to our work includes Thema [21] and Perpetual [26]. Thema is a BFT system for replication of multi-tier Web Services. Perpetual is a BFT system for n-tier and service-oriented architectures, that not only provides replication but also enforces fault isolation. Both are designed for general-purpose Web Services applications and, as such, they lack the mechanisms designed for WS-AT transaction coordination for reducing the runtime overhead as described in this paper.

The other line of system-level research related to our work is that of [27], [32]. Both works address replication of standalone database systems, whereas our work focuses on the trustworthy coordination of distributed transactions and does not address database replication.

The idea of collective determination of the transaction id in our paper is inspired by common practices in security protocols, especially by the work on contributory group key agreement [30], [31].

Finally, like Byzantine quorum systems [19], [20], our BFT framework relies on the use of quorums. However, there are differences. In particular, our BFT Registration protocol requires only $f+1$ non-faulty replicas to receive a Registration request, but it still requires Byzantine agreement on the set of registration records (deferred to the Completion and Distributed Commit protocol). Moreover, by exploiting the semantics of the WS-AT specification, our BFT framework employs Byzantine agreement only where it is needed. Thus, it achieves better performance than if such a Byzantine quorum system were used (naively) to provide trustworthy coordination of Web Services Atomic Transactions.

VII. Conclusion

In this paper, we have addressed the problem of trustworthy coordination of Web Services Atomic Transactions. We have described a suite of protocols and mechanisms that protect the WS-AT services and infrastructure against Byzantine faults. The main contribution of this paper is that it shows how to avoid naively applying a general-purpose BFT algorithm (*i.e.*, totally ordering all incoming requests at the replicated Coordinator), by exploiting the semantics of WS-AT operations to reduce the number of Byzantine agreements needed to achieve atomic termination of a Web Services Atomic Transaction. The cost savings are substantial when the number of Participants is large.

We have incorporated our BFT protocols and mechanisms into an open-source framework that implements the standard WS-AT specification. The augmented WS-AT framework shows only moderate runtime overhead. It outperforms a reference implementation, that naively applies the PBFT algorithm to the WS-AT coordination problem, in both LAN and WAN environments. The augmented WS-AT framework is particularly useful for business applications based on transactional Web Services that require a high degree of dependability, security and trust.

Acknowledgment

An earlier version of this paper was presented at ICSOC'07 [36], and an earlier version of the distributed commit protocol was presented at DASC'07 [37]. This work was supported in part by NSF grant CNS 08-21319 and NSF grant CNS 10-16193, and by a CSUSI grant from Cleveland State University.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, October 2005, 59–74.

[2] Y. Amir, B. A. Coan, J. Kirsch and J. Lane. Byzantine replication under attack. *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, Anchorage, AK, June 2008, 105–114.

[3] Apache Axis project. http://ws.apache.org/axis/

[4] Apache Kandula project. http://ws.apache.org/kandula/

[5] Apache WSS4J project. http://ws.apache.org/wss4j/

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau and J. Cowan, World Wide Web Consortium, *Extensible Markup Language (XML) 1.1*, Second Edition, August 2006.

[7] J. Cowling, D. Myers, B. Liskov, R. Rodrigues and L. Shri. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. *Proceedings of the 7th Symposium on Operating Systems Design and Implementations*, Seattle, WA, November 2006, 177–190.

[8] M. Castro and B. Liskov. Practical Byzantine fault tolerance. *Proceedings of the 3rd Symposium on Operating Systems and Implementation*, New Orleans, LA, February 1999, 173–186.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[10] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin and T. Riche. UpRight cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009, 277–290.

[11] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, World Wide Web Consortium, *Web Services Description Language (WSDL) 1.1*, March 2001.

[12] M. Correia, N. F. Neves, L. C. Lung and P. Verssimo. Worm-IT - A wormhole-based intrusion-tolerant group communication system. *Journal of Systems and Software*, 80(2):178-197, February 2007.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1983.

[14] M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau, H. F. Nielsen, A. Karmarkar, *et al.*, World Wide Web Consortium, *Simple Object Access Protocol (SOAP)*, Version 1.2, April 2007.

[15] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406, November 2001.

[16] R. Kotla, L. Alvisi, M. Dahlin, A. Clement and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007, 45–58.

[17] L. Lamport, R. Shostak and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[18] M. Little and A. Wilkinson. Web Services Atomic Transactions, Version 1.1, OASIS Standard, April 2007.

[19] D. Malkhi and M. Reiter. Byzantine quorum systems. *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, Toulouse, France, October 1997, 569–578.

[20] J. P. Martin, L. Alvisi and M. Dahlin. Small Byzantine quorum systems. *Proceedings of the International Conference on Dependable Systems and Networks*, Washington, D.C., June 2002, 374-383.

[21] M. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware for Web Services applications. *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, Orlando, FL, October 2005, 131–142.

[22] C. Mohan, R. Strong and S. Finkelstein. Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, Canada, August 1983, 89–103.

[23] L. E. Moser and P. M. Melliar-Smith. Byzantine-resistant total ordering algorithms. *Journal of Information and Computation*, 150:75–111, 1999.

[24] L. E. Moser, P. M. Melliar-Smith and N. Narasimhan. The SecureGroup group communication system. *Proceedings of the IEEE Information Survivability Conference* II, Hilton Head, SC, January 2000, 256–279.

[25] S. Northcutt and J. Novak. *Network Intrusion Detection*, 3rd ed., New Riders Publishing, 2002.

[26] S. L. Pallemulle, H. D. Thorvaldsson and K. J. Goldman. Byzantine fault-tolerant Web Services for n-tier and Service Oriented Architectures. *Proceedings of the 28th IEEE International Conference on Distributed Computing Systems*, Beijing, China, June 2008, 260–268.

[27] N. Preguica, R. Rodrigues, C. Honorato and J. Lourenco. Byzantium: Byzantine-fault-tolerant database replication providing snapshot isolation. *Proceedings of the Fourth Workshop on Hot Topics in System Dependability*, San Diego, CA, December 2008, 9.

[28] M. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems*, Lecture Notes in Computer Science 938, Springer-Verlag, 1995, 98-110.

[29] K. Rothermel and S. Pappe. Open commit protocols tolerating commission failures. *ACM Transactions on Database Systems*, 18(2):289–332, June 1993.

[30] M. Steiner, G. Tsudik and M. Waidner. Diffie-Hellman key distribution extended to group communication. *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, New Delhi, India, March 1996, 31-37.

[31] M. Steiner, G. Tsudik, and M. Waidner. CLIQUES: A new approach to group key agreement. *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998, 380.

[32] B. Vandiver, H. Balakrishnan, B. Liskov and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007, 59–72.

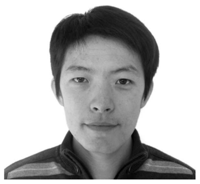
[33] G. S. Veronese, M. Correia, A. B. Bessani and L. C. Lung. Spin one's wheels: Byzantine fault tolerance with a spinning primary. *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*, Niagara Falls, NY, September 2009, 135–144.

[34] The Open Group. *Distributed Transaction Processing: The XA Specification*, February 1992.

[35] A. Young and M. Yung. *Malicious Cryptography: Exposing Cryptovirology*. John Wiley & Sons, New York, NY, 2004.

[36] W. Zhao. Byzantine fault tolerant coordination for Web Services atomic transactions. *Proceedings of the 5th International Conference on Service-Oriented Computing*, Lecture Notes in Computer Science 4749, Springer Verlag, Vienna, Austria, September 2007, 307–318.

[37] W. Zhao. A Byzantine fault tolerant distributed commit protocol. *Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, Columbia, MD, September 2007, 37–44.



Honglei Zhang Mr. Zhang is doctoral student in the Department of Electrical and Computer Engineering at Cleveland State University. He received a Master of Science degree in Electrical Engineering in 2007 from Cleveland State University. His research interests include Byzantine fault tolerance computing and service oriented computing.

Hua Chai Ms. Chai is a doctoral student in the Department of Electrical and Computer Engineering at Cleveland State University (CSU). She received a Master of Science degree in Electrical Engineering in 2009 from CSU. Her research interests include fault tolerance computing, event streaming processing, and service oriented computing.

Wenbing Zhao Dr. Zhao received the Ph.D. degree in Electrical and Computer Engineering from the University of California, Santa Barbara, in 2002. Currently, he is an associate professor in the Department of Electrical and Computer Engineering at Cleveland State University. His current research interests include distributed systems, computer networks, fault tolerance and security. Dr. Zhao has more than 70 academic publications.

P. Michael Melliar-Smith Dr. Melliar-Smith is a professor in the Department of Electrical and Computer Engineering at the University of California, Santa Barbara. Previously, he worked as a research scientist at SRI International in Menlo Park. His research interests encompass the fields of distributed systems and applications, and network architectures and protocols. He has published more than 275 conference and journal publications in computer science and engineering. He has served on many conference program committees, and as a reviewer for many conferences and journals. Dr. Melliar-Smith is a pioneer in the field of fault-tolerant distributed computing. He received a Ph.D. in Computer Science from the University of Cambridge, England.

Louise E. Moser Dr. Moser is a professor in the Department of Electrical and Computer Engineering at the University of California, Santa Barbara. Her research interests span the fields of computer networks, distributed systems and software engineering. Dr. Moser has authored or coauthored more than 260 conference and journal publications. She has served as an associate editor for the IEEE Transactions on Services Computing and the IEEE Transactions on Computers and as an area editor for IEEE Computer magazine in the area of networks. She has served as Technical Co-Chair of the 2011 IEEE International Conference on Web Services and on many conference program committees. She received a Ph.D. in Mathematics from the University of Wisconsin, Madison.