2014

Application Aware for Byzantine Fault Tolerance

Hua Chai
Cleveland State University

APPLICATION AWARE FOR BYZANTINE FAULT TOLERANCE

HUA CHAI

Master of Science in Electrical Engineering

Cleveland State University

12, 2009

submitted in partial fulfillment of requirements for the degree

DOCTOR OF ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

12, 2014

We hereby approve the dissertation
of
Hua Chai

Candidate for the Doctor of Engineering degree.
This dissertation has been approved for the Department of

Electrical and Computer Engineering

and CLEVELAND STATE UNIVERSITY
College of Graduate Studies by

_____

Wenbing Zhao, Dissertation Committee Chairperson J Department & Date

_____

Nigamanth Sridhar, Dissertation Committee Member J Department & Date

_____

Yongjian Fu, Dissertation Committee Member J Department & Date

_____

Lili Dong, Dissertation Committee Member J Department & Date

_____

Haodong Wang, Dissertation Committee Member J Department & Date

11/21/2014

_____

Student's Date of Defense

_____

This student has fulfilled all requirements for the Doctor of Engineering degree.
Dan Simon, Doctoral Program Director

ACKNOWLEDGEMENTS

Many thanks go to my advisor, Dr. Wenbing Zhao, for making me do this dissertation research.

Thank you to my family and friends for standing by me all the time.

APPLICATION AWARE FOR BYZANTINE FAULT TOLERANCE

HUA CHAI

ABSTRACT

Driven by the need for higher reliability of many distributed systems, various replication-based fault tolerance technologies have been widely studied. A prominent technology is Byzantine fault tolerance (BFT). BFT can help achieve high availability and trustworthiness by ensuring replica consistency despite the presence of hardware failures and malicious faults on a small portion of the replicas. However, most state-of-the-art BFT algorithms are designed for generic stateful applications that require the total ordering of all incoming requests and the sequential execution of such requests. In this dissertation research, we recognize that a straightforward application of existing BFT algorithms is often inappropriate for many practical systems: (1) not all incoming requests must be executed sequentially according to some total order and doing so would incur unnecessary (and often prohibitively high) runtime overhead; and (2) a sequential execution of all incoming requests might violate the application semantics and might result in deadlocks for some applications. In the past four and half years of my dissertation research, I have focused on designing lightweight BFT solutions for a number of Web services applications

(including a shopping cart application, an event stream processing application, Web service business activities (WS-BA), and Web service atomic transactions (WS-AT)) by exploiting application semantics. The main research challenge is to identify how to minimize the use of Byzantine agreement steps and enable concurrent execution of requests that are commutable or unrelated. We have shown that the runtime overhead can be significantly reduced by adopting our lightweight solutions. One limitation for our solutions is that it requires intimate knowledge on the application design and implementation, which may be expensive and error-prone to design such BFT solutions on complex applications. Recognizing this limitation, we investigated the use of Conflict-free Replicated Data Types (CRDTs) to construct highly concurrent Byzantine fault tolerance systems, which does not require exploiting too many application semantics since all operations are commutative.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

ACRONYMS

BFT:	Byzantine Fault Tolerance
BA:	Byzantine agreement
PBFT:	Practical Byzantine Fault Tolerance
STM:	Software Transactional Memory
Q/U:	Query/Update
HQ:	Hybrid Quorum
WS-BA:	Web Services Business Activity
WS-BA-I:	Web Services-BusinessActivity-Initiator
WS-RM:	Web Services Reliable Messaging
WS-AT :	Web Services Atomic Transactions
BAwPC:	BusinessAgreement-with-Participant-Completion
BAwCC:	BusinessAgreement-with-Coordinator-Completion
2PC:	Two Phase Commit
EPA:	Event processing agent
CRDT:	Conflict-free Replicated Data Type

CHAPTER I

INTRODUCTION

Nowadays, distributed systems are playing an ever more important role in business operations. Considering the untrusted operating environment of the Internet, such applications are often required to be highly available and trustworthy for being protected against both crash and malicious faults. A promising technology that could help achieve high availability and trustworthiness is Byzantine fault tolerance (BFT). BFT provides correct services and high availability through redundancy, which is deploying replicated servers, despite the presence of Byzantine faulty replicas and clients in the system. In the past decade, we have seen tremendous efforts to bring BFT to distributed systems. However, we have yet to see widespread adoption of the BFT technology in practice. It is well known that BFT can be achieved by the use of a BFT algorithm to ensure that all non-faulty replicas reach an agreement, referred to as a Byzantine agreement (BA), on the total ordering of requests to a server. However, most state-of-the-art BFT algorithms are designed for general purpose stateful applications and often incur significant runtime overhead. In addition, the system models of practical applications (such as business process applications) are often much more complicated than those used in the BFT algorithms.

We recognize that many distributed Web-based application systems are designed to be session-oriented and messages in those systems can be partitioned based on the state they are modifying. Those messages that belong to different partitions can be considered commutative and executed in parallel [1]. This investigation leads us to design BFT frameworks for such systems by considering their application semantics, which we refer to in this dissertation as application aware BFT. The design objective of application aware BFT is the same as that of traditional BFT. However, our application aware BFT can significantly reduce the runtime overhead since we do not need to total order all messages at the server when it is not necessary and we allow concurrent processing of requests whenever the semantics allows. We have investigated our approach for several types of distributed Web-based applications e.g., session-oriented shopping cart, event stream processing, Web Services Business Activities (WS-BA) [2] and Web Services Atomic Transactions (WS-AT) [3], and designed and implemented a lightweight BFT framework for each of them. Performance evaluation of the prototype implementation of our BFT frameworks confirms the efficiency and effectiveness of this lightweight approach. The only limitation of the application aware BFT is that it requires intimate knowledge on the application design and implementation, so the development cost might be high for complex applications.

Recent research also suggests that a service may be constructed by using Conflict-free Replicated Data Types (also referred to as commutative or convergent replicated data types, or CRDTs in short) for highly concurrent optimistic replication with the crash-fault model [4-7]. In this dissertation research, we extend such studies by adopting the Byzantine fault model [8], which encompasses both crash faults and malicious faults. We carefully

analyze the threats towards the operations in a replicated service constructed with CRDTs, and propose a lightweight solution to achieve optimistic BFT. We can execute all independent requests in parallel without total ordering of these requests since the operations for these requests are commutative. The primary challenge of our solution is to handle an insidious threat imposed by an adversary: it may disseminate conflicting information to different replicas. We mitigate such threats and ensure eventual Byzantine consistency by engaging in on demand and periodic synchronization of the replica states via a BA service. A round of state synchronization can be triggered on-demand by a client upon detecting the presence of conflicting requests, or periodically when the replica has executed a predefined number of requests. Performance and evaluation results show that our solution is efficient and with low runtime overhead.

CHAPTER II

LITERATURE REVIEW

This chapter provides an overview of the background of this dissertation research. Byzantine fault, BFT and related topics are elaborated.

2.1 Background

2.1.1 Byzantine Fault

A Byzantine fault [9] is an arbitrary fault, which could be a crash or malicious fault. The occurrence of such a fault may cause a system to enter into an arbitrary failure state. The failure behavior could be: crashing, failing to receive a request or processing a request incorrectly, failing to send a response or sending an incorrect/inconsistent response, or corrupting local state. When a Byzantine fault occurs, it is hard to be detected and the system's state and outcome may differ significantly from that under normal execution.

The term “Byzantine fault” is originated from the study on the Byzantine agreement problem by Marshall Pease, Robert Shostak, and Leslie Lamport in 1980 [10]. The problem is framed in a scenario that the Byzantine Empire's army needs to decide whether to attack enemy armies by sending messages to each other. By the presence of traitors, some generals in the Byzantine Empire's army may get messages to attack while others may get different messages to fail to make up their minds. The non-faulty generals must have a unanimous agreement on their strategy. The commanding general could be correct or incorrect. In either case, all non-faulty generals must agree upon the same decision.

2.1.2 Byzantine Fault Tolerance

BFT is the capacity of a system to tolerate Byzantine faults, which can be achieved through replicating application servers and executing BFT algorithms. A lot of BFT algorithms can achieve fault tolerance by ensuring that all non-faulty replicas reach an agreement, referred to as Byzantine agreement (BA), on the total ordering of requests to a server despite the presence of Byzantine faulty replicas and clients in the system. A system using such a BFT algorithm requires that all application requests are executed sequentially to ensure state consistency among replicas which imposes a big limitation on the performance of the systems. There are some classic BFT algorithms of such type but with different replicating sizes. BFT algorithms, such as Practical Byzantine Fault Tolerance (PBFT) [8] and Zyzzyva [11], typically require the use of 3f + 1 server replicas where at most f replicas are faulty. It means even if all f replicas are faulty they still can ensure both safety and liveness of the system. Other BFT algorithms may require different replication degrees, such as Q/U (Query/Update) [12] and HQ (Hybrid Quorum) [13]. Q/U requires 5f

+ 1 server replicas, where at most f replicas are faulty. HQ uses combined approaches that it uses 2f + 1 server replicas when there is no conflict detected; otherwise, it uses 3f + 1 server replicas.

2.1.3 Web Services Business Activities

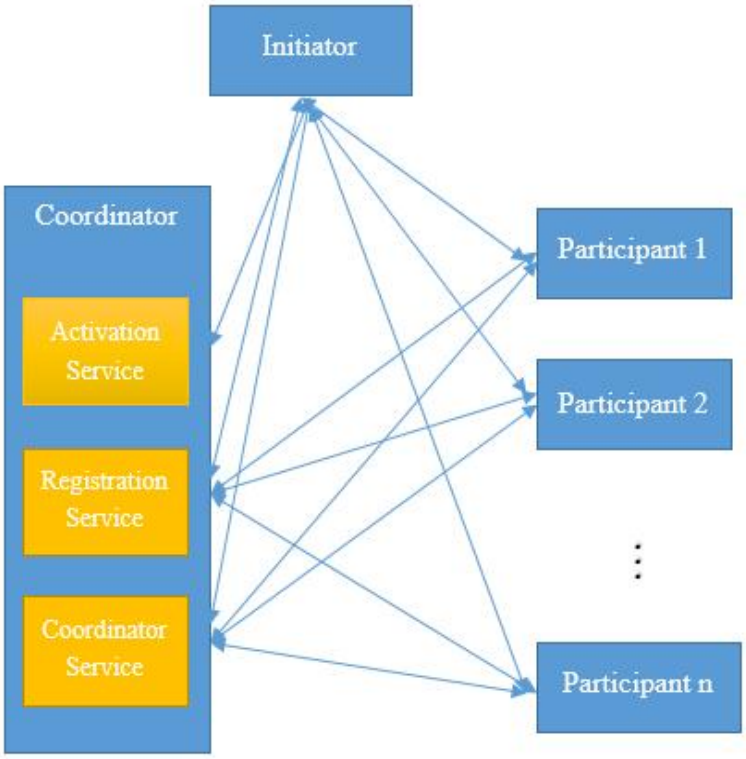


Figure 1 WS-BA components

The WS-BA specification [2], developed by OASIS, defines a set of services to coordinate entities (the Initiator, the Coordinator and one or more Participants) involved in a Business Activity, shown in Figure 1. The Coordinator run the Activation service, the Registration service and the Coordinator service in the same address space of the

Coordinator. The Initiator requests the Activation service to generate a Coordinator object and a Coordination context with a unique Coordination identifier for each new Business Activity. The Coordinator object provides the endpoint references for Participants to access the Registration and Coordinator services.

WS-BA is built on top of the WS-Coordination framework [12] and specifies two Coordination types: Atomic-Outcome and Mixed-Outcome. With Atomic-Outcome Coordination, all Participants must agree on the outcome of the Business Activity (i.e., Close or Compensate). With Mixed-Outcome Coordination, it allows that some Participants are directed to close while others are directed to compensate. Also, WS-BA specifies two Coordination protocols: BusinessAgreement-with-Coordinator-Completion (BAwCC) and Business-Agreement-with-Participant-Completion (BAwPC). Either Coordination type can be combined with using either Coordination protocol. We briefly describe the two Coordination protocols below.

BAwPC protocol. When a Participant of a Business Activity has finished its processing, it notifies the Coordinator by sending a Completed message. If the Business Activity has completed successfully, the Coordinator sends the Participant a Close message; otherwise, it sends the Participant a Compensate message to undo its change. If the Participant fails to complete a Business activity, it sends a Fail message to the Coordinator. Similarly, if the Participant fails to cancel or compensate a Business activity, it sends a CannotComplete message to the Coordinator.

BAwCC protocol. The completion notification comes from the Coordinator instead of the Participants. The Coordinator sends a Complete message to the Participants, informing them to complete their processing of the Business Activity.

The Coordinator service interacts with the Initiator via the WS-BA-I protocol [15]. The WS-BA-I is an extension of the WS-BA. In the WS-BA-I protocol, the Initiator can query the state of the Business Activity and create invitation tickets. The Initiator can query the latest status of each Participant from the Coordinator so that it can make decision to Complete (only for the BAwCC protocol), Close, Cancel or Compensate the Business Activity. The Initiator also can ask the Coordinator to create invitation tickets which can be used to propagate the Business Activity to a remote Participant Web Service. Each invitation ticket is an identifier used to invite a Participant to join the Business Activity. The remote Participant Web Service then uses this ticket to register with the Coordinator.

2.1.4 Web Services Atomic Transactions

The WS-AT specification [3] is one of the Web Services standards adopted by OASIS. The standard specifies a set of services and two protocols: the two Phase Commit (2PC) protocol and the Completion protocol. The Completion protocol is run between the Initiator and the Coordinator and the 2PC protocol [16] is run between the Coordinator and the Participants. These protocols and services together ensure automatic activation, registration, propagation, and atomic termination of a distributed transaction based on Web Services, similar to the example shown in Figure 1.

The Coordinator-side has the following services:

• Activation Service: The Activation service creates a transaction context and a Coordinator object for each transaction. The Coordinator object provides the Registration service, the Completion service and the Coordinator service. The transaction context

contains a unique transaction identifier and an endpoint reference for the Registration service, and is included in all request messages sent during the transaction.

• Registration Service: The Registration service allows the Participants and the Initiator to register their endpoint references.

• Completion Service: As part of the Completion protocol, the Initiator calls the Completion service to start the distributed commit process.

• Coordinator Service: This service runs the 2PC protocol to ensure atomic commitment of the distributed transaction.

The Initiator offers the Completion Initiator Service, which is used by the Coordinator to inform the Initiator of the final outcome of the transaction, as part of the Completion protocol.

The Participant offers the Participant Service, which is used by the Coordinator to solicit votes from, and to send the transaction outcome to, the Participant.

The 2PC protocol commits a transaction in the following two phases: Prepare phase and Commit/Abort phase. During the Prepare phase, the Coordinator sends a Prepare request to all Participants for them to prepare to commit the distributed transaction. Once received a Prepare request, a Participant must be prepared to either commit or abort the transaction. A Participant prepares the transaction for commitment and responds with a Prepared vote if it can commit the transaction; otherwise, it responds the Coordinator with an Abort vote. A Participant that has not responded with a Prepared vote can have the Coordinator 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

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 Prepare phase; otherwise, it decides to abort the transaction.

2.1.5 Session-Oriented Multi-tier Applications

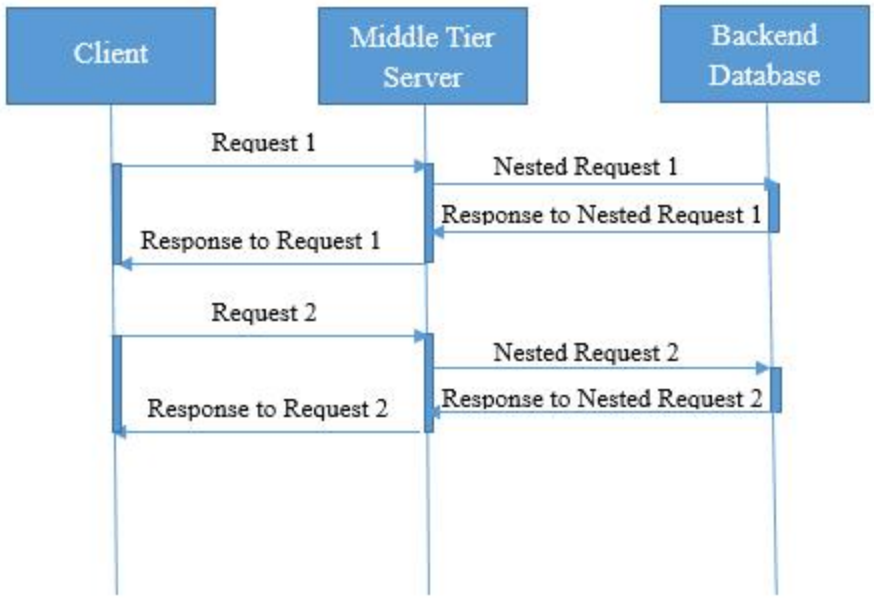


Figure 2 Typical interactions in a three-tier application

Many Web-based applications are organized according to the multi-tiered architecture, where the client communicates directly with the middle-tier server, and the middle-tier server in turn interacts with the backend server for persistent state when needed, as shown in Figure 2.

When a client first invokes a service in a Web-based application, the service instance at the middle tier creates a session for this client. All messages exchanged within this session will carry a unique session identifier. With the session identifier as a reference,

the client can operate on its session state with a sequence of operations through the service. To process an invocation from the client, the service may issue nested invocations on the backend database server. When the session is over, the service closes the session for the client and removes non-persistent data stored for the session.

Different sessions may share state, but only through the backend servers. Hence, the requests that belong to different sessions can be handled in parallel. The execution of conflicting requests sent by a faulty client would only impact the client itself and the corresponding session state, which is limited to this particular session.

The backend server is used to store persistent data. While persistent data is stored at the backend server, the middle-tier server often does maintain session state. The execution in different sessions will be synchronized at the backend server if some state is shared among these sessions.

2.1.6 Event Stream Processing

Event stream processing is a powerful way of constructing distributed systems that take input continuously from various sources (referred to as event producers), and generate alerts and derived events according to predefined rules for consumption by their clients (referred to as event consumers). Event stream processing can be regarded as a variation of message-oriented middleware technology, and it has been used in many mission-critical applications, such as business intelligence and collaborative intrusion detection [17, 18].

An event stream processing system consists of three major components, the event producers, one or more event processing agents (EPAs), and the event consumers, as shown

in Figure 3. Event producers are the entities that produce events (e.g., sensors). The brain of an EPA is determined by a set of rules. The rules are typically predefined and they may be updated dynamically. Event consumers are entities that are interested in the derived events or alerts generated by an EPA.

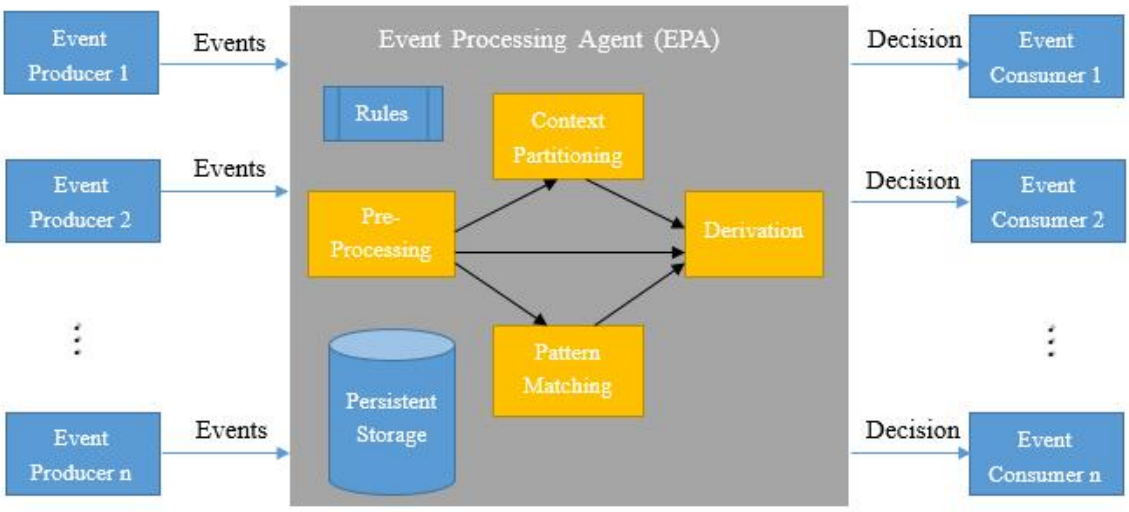


Figure 3 An event stream processing system for autonomic computing

The EPA is usually powered by specialized event stream processing middleware frameworks such as Esper [19], which simplifies the design and implementation of autonomic components, as shown in Figure 3. The EPA processes incoming events according to the execution rules. Normally, incoming events are first pre-processed according to predefined criteria and non-qualified events are discarded. The middleware framework typically provides either user interfaces or application programming interfaces to revise the processing rules dynamically. New events can be derived from pre-processed events. As shown in Figure 3, the derivation step occurs right after the pre-processing step. During the derivation step, an event might be translated, enriched, or projected into another event for the event consumer to handle [20]. Event processing could be stateless or stateful.

For stateless processing, each event is processed independently from other events. More complex event processing is stateful in that multiple events are considered collectively. Stateful processing usually belongs to one of the following two types:

- Aggregation oriented: Attributes of a sequence of events possibly belong to different event streams are computed to derive higher level information.
- Pattern matching oriented: Predefined event patterns are being matched against the incoming events.

For stateful event processing, typically the operation is applied to a subset of the events in one or multiple streams. The subset is determined by the context as defined in the processing rule. The events may be partitioned for context formation based on a number of criteria:

- Time: The context is determined based on the timestamps of the events. Events that belong to one or more time intervals may belong to a temporal context. Different temporal contexts may overlap. In fact, sliding windows are one of the most common temporal contexts used in complex event processing.
- Location: The context is determined based on the location where events occurred.
- Identifier: Events may contain an identification at- tribute, such as customer id, that can be used to partition the events to different contexts.
- State: Which events are included in a context depends on the external state. The external state may in turn be temporal, spatial, or identifier based.

The communication pattern between different components of an event stream processing system is drastically different from that of client- server systems. In client-server systems, the client typically sends requests to the server for processing, and blocks waiting for the corresponding replies before it issues the next one. In event stream processing systems, on the other hand, different components communicate via one-way messages:

- An event producer continuously streams events to the EPA and does not expect any reply.
- When an event consumer wants to update or create a new processing rule, it sends a one-way request to the EPA.
- The EPA sends an alert message to the designated event consumer whenever one is generated at the end of the derivation step.

2.1.7 Conflict Free Data Replicated Types

CRDTs are also referred to as commutative or convergent replicated data types. Shapiro et al., [7] first proposed to use CRDTs to build replicated services since operations towards the system constructed with one or more CRDTs are considered commutative. In [7], a list of CRDTs are also introduced including various types of counter, registers, sets, maps, graphs, and sequences.

A replicated system constructed using CRDTs can avoid the complexity of conflict resolution and of roll-back with the presence of crash faults. For such a system, an update does not require synchronization and the state will be eventually converged after temporary

network partitioning is healed. Constructing system using CRDTs can be trivial or non-trivial depending on the specific replicated data type. E.g., replicating counter is trivial since it only has increment and decrement operations and they commute. A non-CRDT construct needs to be converted to a CRDT construct before replication. The well-known example for this case in [7] is converting the set data type to U-set, which still can be considered trivial. There are also a number of non-trivial CRDTs such as Treedoc, which is a sequence CRDT for co-operative text editing [5].

2.2 Related Work

2.2.1 State Machine Oriented BFT Algorithms

BFT has been of great research interest for the last several decades. The seminal research in this field is that of Lamport, Shostak and Pease [9], who demonstrated that four replicas are required to tolerate one Byzantine faulty replica (three replicas do not suffice) and, more generally, that 3f + 1 replicas are required to tolerate f Byzantine faulty replicas. There exists a number of general-purpose BFT algorithms/protocols [8, 11-13, 21, 22] and group communication systems that handle Byzantine faults [23-27] for generic distributed applications. All of those systems deliver messages in total order at the destinations. In contrast, our lightweight BFT frameworks discussed in this dissertation do not require total ordering all incoming messages at replicated servers when total ordering is not necessary.

2.2.2 Web Services Atomic Transactions

The application of BFT techniques to transactional systems was first reported by Mohan et al. [28]. They enhance the 2PC protocol by performing Byzantine agreement (BA) on the outcome of an atomic transaction among all of the nodes in a root cluster. This problem has been revisited and a more efficient solution is proposed [29] for Atomic Transactions by restricting the BA to only the Coordinator replicas. Garcia-Molina et al. [30] have applied BA to distributed database systems, in particular, for distributing transactions to processing nodes.

The system-level research [32, 33] is closest to our work [31]. Perpetual [33] is a BFT system for n-tier and service-oriented architectures that provide replication and also enforces fault isolation. Thema [32] is a BFT system for replication of multi-tiered Web Services. Both are designed for general-purpose Web Services applications but they do not have the mechanisms designed for WS-AT transaction coordination for reducing the runtime overhead. The other system-level research related to this work is [34, 35] addressing replication of standalone database systems, whereas our work focuses on the trustworthy coordination of distributed transactions instead of database replication.

Our BFT framework [31] for the WS-AT services relies on the use of quorums like Byzantine quorum systems [36, 37]. However, the Registration protocol in our BFT framework requires only f + 1 non-faulty replicas but it still requires a BA on the set of registration records. By exploiting the semantics of applications implementing the WS-AT specification, our BFT framework employs BA only where it is needed. Thus, to provide

trustworthy coordination of Web Services Atomic Transactions, our BFT framework achieves better performance than that of a Byzantine quorum system.

In our BFT framework [31], the solution of collective determination of the transaction identifier inspired by common practices in security protocols, especially by contributory group key agreement [38, 39].

2.2.3 Web Services Business Activities

The solutions proposed in [32, 33, 40] could be used to protect the Coordination services of Web Services Business Activities against Byzantine faults but they are unnecessarily expensive. By considering the state model and exploiting the semantics of the WS-BA Coordination services, our lightweight BFT algorithm [41] for the WS-BA services introduces only one additional round of message exchange for each invocation on the Coordinator instead of two or more additional rounds of message exchange needed in [32, 33, 40] and, thus, it performs significantly better.

2.2.4 Event Streaming Processing

A crash-fault tolerance solution for event stream processing was proposed in [42] with active replication of EPA. The proposed solution is specifically designed for systems constructed using software-transactional memory [43]. The most interesting mechanism is to allow concurrent processing of events by exploiting the transaction processing model. Specifically, a validation step is introduced prior to the commit of each transaction to

ensure that transactions are committed according to the total order of the events that triggered the transactions.

Previously, Zhang et al. extended the work in [42] to tolerate Byzantine faulty EPA replicas by using a BFT algorithm [8] to totally order all event messages in [44]. Although similar in objective, this dissertation work [45] takes a completely different approach. First, we do not rely on the software-transactional memory processing model. Second, by exploiting the semantics of event stream processing for autonomic processing, we designed an on-demand state synchronization mechanism to ensure the consistency of EPA replicas and the decisions sent to event consumers.

Fault tolerant autonomic computing has been studied by several researchers [46] [17]. However, the solutions proposed are restricted by the crash-fault model used and they cannot survive malicious attacks.

In general, application-aware BFT is a promising research direction because it gives guidelines on how to develop BFT solutions for practical systems by using basic Byzantine agreement (BA) constructs in a way similar to designing secure systems by using cryptography primitives [1, 31, 41, 45, 47]. The research in this dissertation belongs to this line of work.

In [1], the semantics of the network file system is utilized to parallelize the execution of non-conflicting requests. However, to ensure correct partial ordering of conflicting requests, all requests have to be totally ordered in the first place, which may incur additional latency. In [48], a further improvement is proposed by executing at a subset of the replicas but all requests still must be totally ordered in the first place, which is different from our approach [31, 41, 46, 47].

2.2.5 Conflict Free Data Replicated Types

Optimistic replication [49], where a replica is allowed to execute a client's request immediately, without block waiting until the request is totally ordered, is an attractive approach to practitioners due to its low runtime overhead and robustness under network partitioning faults compared to traditional conservative approaches. This is the case as seen from the development of the CAP theorem [50] and the optimistic replication strategy employed in practical cloud systems [51].

Optimistic replication aims to achieve eventual consistency among the replicas [49], i.e., the states of the replicas will eventually converge when the clients stop issuing new requests and the network is reasonably connected so that a request executed at one replica can be propagated to all other non-faulty replicas. However, enabling concurrent executions of potentially conflicting requests is challenging. Operational transformation was proposed to facilitate the convergence of the states of different replicas involved with conflicting operations [52]. However, it has been pointed out in [53] that most operational transformation algorithms for decentralized architecture are not correct.

If updates are applied concurrently, it may conflict and may require a consensus and a roll-back. However, conflict resolution is hard and ad-hoc approaches are error-prone. A typical example is the concurrency anomalies of the Amazon Shopping Cart [54]. To solve the issue with a simple approach, Shapiro et al. proposed to use CRDTs to build replicated services [7] under the crash-fault model. In [7], Shapiro et al also introduced a list of CRDTs including various types of counter, registers, sets, maps, graphs, and sequences. It is shown that these CRDTs can be used to build practical applications.

However, this work only handles crash faults, which is hard to be extended to handle malicious faults if the system has a faulty entity, but it helps prove that with using CRDTs we do not need to totally order all requests so avoid suffering from the problems related to operational transformations.

CHAPTER III

BFT FOR WEB SERVICES BUSINESS ACTIVITIES

In this chapter, we use a travel reservation application as a running example, to illustrate the lightweight BFT framework we designed for protecting the integrity of the WS-BA coordinator services.

The travel reservation application, designed with using the WS-BA specification and the WS-BA-I extension [15], is a composite Web Service which consists of an Initiator, a Coordinator, a Car Rental Web Service and a Hotel Reservation Web Service. Both the Car Rental Service and the Hotel Reservation service contain a Participant service. The Coordinator is also a composite Web Service which consists of an Activation service, a Registration service and a Coordination service. The travel reservation application is set to use the Atomic-Outcome Coordination type and the Business Agreement with Coordinator Completion (BAwCC) protocol. The role of each service is explained in section 2.1.3. The complete steps of the Business Activity booking cars and hotels by this travel reservation application is shown in detail in Figure 4.

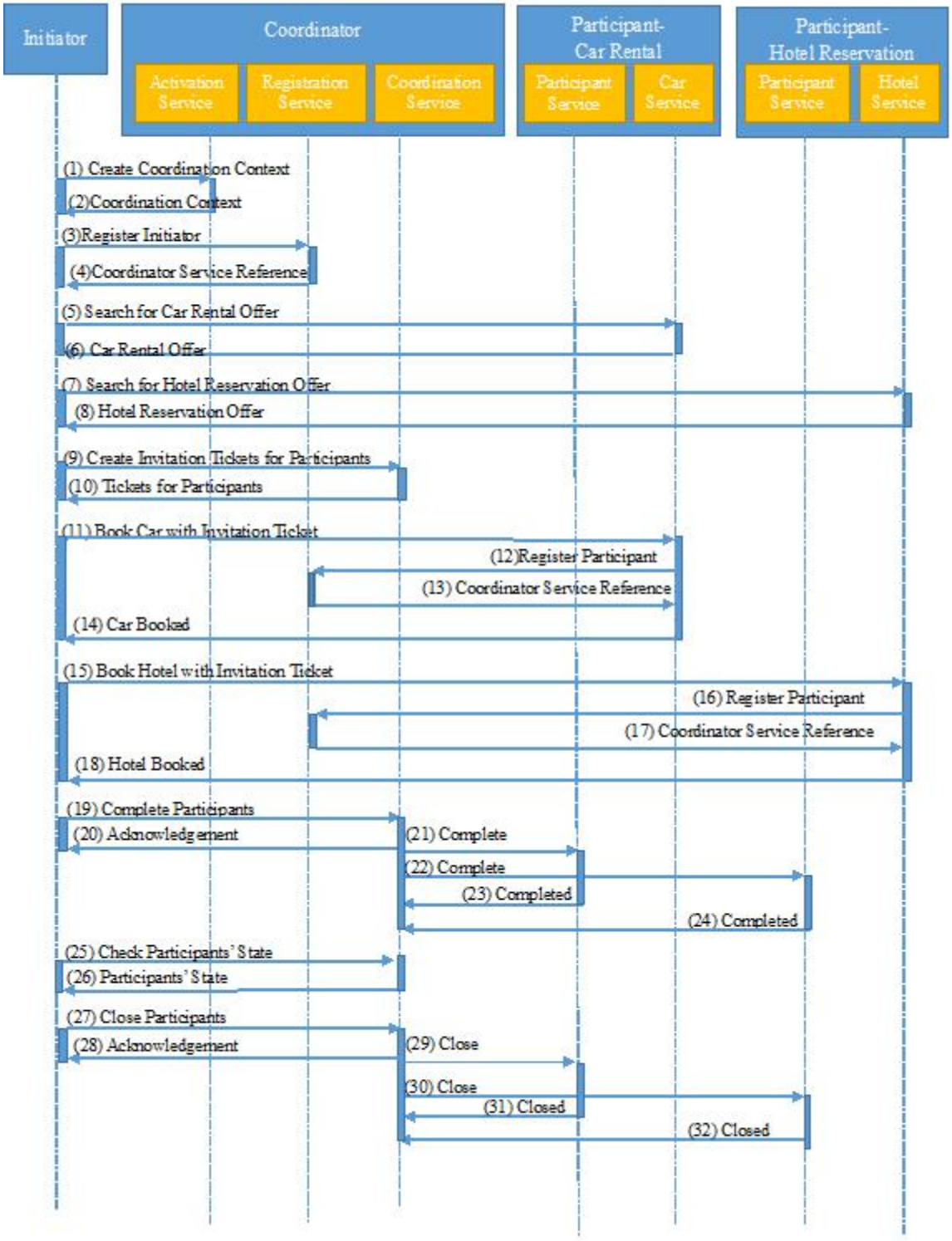


Figure 4 The sequence diagram for a travel reservation application using the WS-BA protocol.

3.1 Threat Analysis

In this section, we analyze threats that can compromise the integrity of the WS-BA Coordination services. We do not consider general threats like service denial attacks since such threats are not in our research scope.

3.1.1 Threats from a Faulty Coordinator

First, we consider threats from the Activation service, the Registration service and the Coordinator service comprised by a faulty Coordinator.

Threats to the Activation Service. The Initiator of a Business Activity requests the Activation service to generate a Coordination context with a Coordination identifier. Among the rest of the process of the Business Activity, the generated Coordination identifier will be used uniquely identify the activity and shared among the Initiator and the Participants.

A faulty Coordinator could use an invalid Coordination identifier for a new Business Activity. The Coordination identifier could be old, easy-to-predict or belonging to another Business Activity.

A faulty Coordinator could use an old Coordination identifier for a new Business Activity. A malicious Participant that is involved in an-already-closed Business Activity could use the previously assigned (old) identifier to eavesdrop the messages sent for the new Business Activity. Such a threat can be mitigated by using a nonce or timestamp.

A faulty Coordinator could use a predictable Coordination identifier. Any uninvited adversary knowing this identifier could use the identifier register with the Business Activity. If the Initiator cannot spot and exclude the adversary from the Business Activity, we can use our lightweight BFT framework to handle this threat.

A faulty Coordinator could use a Coordination identifier for multiple different business activities. If the Coordinator identifier is a used one, it can be eliminated by using nonce or timestamp. Otherwise, this is a real threat and needs to be mitigated, because using the same Coordination identifier could treat two unrelated business activities as one activity, which might confuse the Initiator and the Participants. E.g., one Business Activity b1 is about to close after all its Participants completed their work, at the meantime, another Business Activity b2 just gets its coordination context created. The initiator will encounter a failure later when it tries to register or access the Coordination service for b2 since b2 will be treated as closed. The Participants in the scope of the Business Activity b2 could encounter similar situation. E.g., they could be notified to complete their work even if they has not yet taken any action such as booking a car or hotel room. It could also happen that after a Participant which belongs to the Business Activity b2 just booked a car or hotel room, it receives the Complete notification message and complete its work before the Initiator realizes it might want to abort the whole process for the Business Activity b2. We designed a lightweight BFT algorithm described in the next section to mitigate this threat. Our solution guarantees that for each Business Activity there is at most one unique coordination identifier can be assigned to the Business Activity and accepted by the Initiator and the Participants.

Threats to the Registration Service. A Participant can register with the Registration service by providing an endpoint of its Participant service. The Participant needs to provide the Registration service with a valid Coordination identifier and a matchcode (see in WS-BA-I) for authentication. The matchcode is used for identifying a valid or contracted Participant. The Participant will receive an endpoint reference to the Coordinator's Coordination service if the registration is successful.

A faulty Coordinator could register any unauthenticated entity with an illegitimate credential as a Participant. A faulty Coordinator could also register an authenticated Participant with an invalid endpoint reference. For example, a faulty Coordinator has both a non-reputable hotel reservation company and a reputable hotel reservation company and always have the non-reputable one steal the business by denying the reputable one' participation.

These threats cannot be easily mitigated using transport-level security mechanisms and we resolve them by our lightweight BFT algorithm described in section 3.2.3.

Threats to the Coordinator Service. The Coordinator service interacts with the Initiator via the WS-BA-I protocol and interacts with the Participants via the BAwCC (or BAwPC) protocol. The Coordinator notifies the Participants about the completion of a Business Activity, receives acknowledgement messages and notifications from them, and notifies them the outcome of the Business Activity.

A faulty Coordinator could notify Participants Complete, Close or Compensate their work without the permission of the Initiator. Such a thread can be mitigated by requiring all notification messages to carry a security token [29] which could be a signed certificate for the action from the Initiator.

A faulty Coordinator could also make arbitrary state transitions no matter whether it receives reports (e.g., Fail or CannotComplete messages) from the Participants. Such kind of threats would lead to inconsistent state among the Coordinator and the Participants, which might ultimately affect the outcome of the Business Activity since the Initiator may decide to close Participants without the awareness that some of them may fail to complete their work and need compensation. However, we can handle such threats by replicating the Coordinator and using the lightweight BFT algorithm described in section 3.2.3.

A faulty Coordinator could return an invalid invitation ticket to the Initiator. An invalid invitation ticket could be old, easy-to-predict or belonging to another Participant. The threat caused by using an old invitation ticket can be mitigated by using a nonce or timestamp. However, an adversary can register a faulty Coordinator's Coordination service with an easy-to-predict invitation ticket and join the Business Activity. The adversary may not be able to affect the Initiator's decision if the Coordination service does not lie about the authenticated Participants' state, but the adversary still can eavesdrop all the decision from the Initiator. If an adversary wants to register a faulty Coordinator's the Coordination service with a ticket belonging to another Participant, it even does not need to obtain the Coordination identifier of the targeted Business Activity to authenticate itself since a faulty Coordinator could either share the Coordination identifier with the adversary or just have the adversary skip the authentication. Also, a faulty Coordinator may send the Initiator an invitation ticket belonging to another Participant. If this invitation ticket is created for another Business Activity, reusing such an invitation ticket in this case just increases the risk that an adversary can pass the authentication. However, a faulty Coordinator may even not need an authentication from the adversary. If the invitation ticket

is created for the same Business Activity, then one or both of the Participants could be rejected from the registration while a faulty Coordinator could lie that both Participants have registered successful. We can mitigate these threats by the lightweight BFT algorithm described in section 3.2.3.

3.1.2 Threats from a Faulty Participant

A faulty Participant could try to compromise the integrity of the Coordinator and the Initiator.

Threats to the Coordinator. A faulty Participant could either (1) lie about its execution status to the Coordinator or (2) send conflicting reports to different Coordinator replicas. E.g., for case (1), a faulty Participant could report to the Coordinator the Fail status to prevent the completion of the Business Activity even if the Participant completed its work. For case (2), a faulty Participant could report to a Coordinator replica the Completed status and report to another Coordinator replica the Fail status to have them make different decisions (one to Complete and another to Compensate) and affect the outcome of the Business Activity. To prevent a faulty Participant send reports of the state inconsistent with its internal state in case (1), each Participant can be replicated and up to f of them can be faulty so the Coordinator can accept a report by collecting at least f + 1 matching ones from distinct replicas. However, replicating each Participant in real life is not practical. Therefore, we'd rather address this type of threat by signing each message sent between the Participants and the Coordinator and logging those messages at the Coordinator. The faulty Participant could also prevent the completion of the Business Activity by sending conflicting messages to different Coordinator replicas. To prevent bad consequence caused

by conflicting reports sent by a faulty Participant in case (2), we can address this type of threats by the lightweight BFT algorithm described in section 3.2.3.

Threats to the Initiator. A faulty Participant could attack the Initiator by lying about its execution status to the Initiator. E.g., a faulty Participant could lie to the Initiator that it fails to book a car to prevent the completion of the Business Activity. This type of threats can be addressed by using security tokens on messages exchanged with the Participants and having the Initiator's logging these messages. To minimize such threats, the Initiator could give preference to the Participants that offer a higher quality of service.

3.1.3 Threats from a Faulty Initiator

The integrity of a Business Activity cannot be guaranteed if the Business Activity is started at a faulty Initiator since the faulty Initiator could ignore messages from other entities, send arbitrary messages to other entities, or make decision not based on the responses from other entities during the entire lifecycle of the Business Activity. Such threats can be tolerated by replicating the Initiator. We only handle such type of threats by using digital signatures on messages exchanged between the Initiator and other entities and logging them at the Participants and the Coordinator. However, the Initiator can be easily replicated when this is necessary.

As the Coordinator is replicated, a faulty Initiator may send conflicting requests to distinct Coordinator replicas. Such a threat can be handled by the lightweight BFT algorithm described in section 3.2.3.

3.2 System Model and Solution Design

In this section, we present the lightweight BFT algorithm and associated BFT mechanisms to enable an independent third party to launch trustworthy Coordination services for Web Services Business Activities. In our lightweight BFT algorithm, we exploited the semantics of the WS-BA application by analyzing the state model of its Coordination services and designed a solution that is customized to this application.

3.2.1 System Model Analysis

According to WS-BA-I specification, different Business Activities are independent, the relative ordering of how they get created and processed doesn't affect their outcome.

(1)Requests to the Activation services are independent since their relative ordering does not affect how the Coordinator objects created.

(2)Request to the Registration and Coordination within different Business Activities are independent since they are handled by different Coordinator objects.

Therefore, requests within different Business Activities are independent, so the requests can be handled in parallel, not necessary to be totally ordered.

Requests within the same Business Activity do not need ordered apparently to guarantee the outcome.

(1) Requests to the Activation and Registration services within the same Business Activity do not need ordered apparently since they are already causally related. Based on

the WS-BA-I specification, the request to the Activation services has to precede the request to the Registration service, which needs no effort to guarantee this relationship.

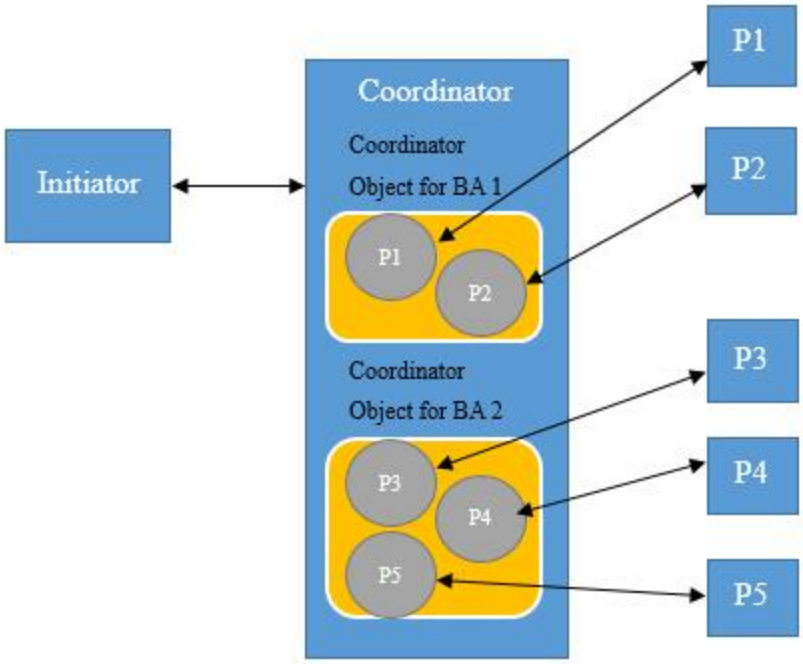


Figure 5 The completely separate states of the Participants in the Coordinator object.

(2) Requests from different Participants to the Coordination services within the same Business Activity do not need ordered apparently. According to the BAwCC (or BAwPC) protocol, a Coordinator object state can be further partitioned into independent sub-states for distinct Participants and each Participant only can change the state associated with itself. Thus, the Participants' states in the Coordinator object are completely separate. The change of one state associated with one Participant won't affect another state associated with a different Participant, as shown in Figure 5.

(3) Requests from the Initiator to the Participant services within the same Business Activity do not need ordered apparently according to the WS-BA-I specification:

- Requests that create invitation tickets must precede the requests from the Participants within the same Business Activity since the Participants have to get the invitation tickets before able to register with and access to the Coordination service.
- Requests that queries the state of the Coordination object of the Business Activity are not ordered with messages from the Participants. These requests are read-only and won't change any state of the Coordination object, but if they are not ordered with the messages from the Participants, it could happen that different Coordinator replicas might send query results containing different states to the Initiator. However, if the Initiator can repeatedly query the Coordinator replicas until their states converge, this should not be a concern anymore.
- Requests that ask the Coordinator to Close, Cancel or Compensate the Business Activity are not ordered with messages from the Participants. It could happen that states in different Coordinator replicas are inconsistent after processed these requests and messages in different orders. However, this should not be a concern. According to the WS-BA-I protocol, when the last observed state for that Participant is Completing, the Initiator could send a Cancel request to the Coordinator for the Participant to cancel its work. The Completed message from the Participant may arrive at the Coordinator earlier than the Cancel request does. In this case, the Coordinator should send a Compensate request to the Participant instead of a Cancel request. The Coordinator sends a Cancel request to a Participant only when the Participant has not completed its work or the Participant's Completed message has not arrived yet. This could cause the Coordinator's state for the Participant ended with either Compensated or Cancelled. However, in both

cases, the order of the requests and messages won't affect that the Participant's application state isn't changed. As the Coordinator is replicated, it could happen one replica send a Cancel request while another send a Compensate request, which may need the Participant voting to select one request to execute. However, there is still no need to ensure total ordering of these requests from the Initiator and messages from the Participants involved in the same Business Activity at the Coordinator replicas.

3.2.2 Assumptions and Requirements

We made the following assumptions and requirements about our system model.

We assume the Web Services Business Activities run in an asynchronous distributed environment with reliable network communication. This can be easily satisfied by using TCP communication and the mechanisms in the Web Services Reliable Messaging (WS-RM) specification [55], which can be adopted to guarantee that each message sent by a non-faulty WS-BA entity (e.g., the Initiator, a Coordinator replica or a Participant) will be eventually received at its receiver.

We assume the Coordinator is replicated with $3f + 1$ replicas and at most f of them are faulty. Unlike most popular traditional BFT algorithms like PBFT [8] and Zyzzyva [11], none of the Coordinator replicas is chosen as the primary and there is no need of total ordering for incoming messages at each replica.

We assume the Participants and the Initiator are not replicated. As pointed out earlier in section 3.1.2 and 3.1.3, imposing such a strong requirement on the Participants

and the Initiator is not practical. Therefore, we only handle such type of threats by using digital signatures on messages exchanged between the Participants/Initiator and other entities and logging them at each Coordinator replica.

We assume all requests have a unique identifier and the identifier could be a sequence number which is monotonically increased between each pair of the WS-BA communication entities. A response message contains the identifier that matches that of its corresponding request.

We assume all messages between the Participants and the Coordinator are digitally signed and timestamped. Including timestamp in each message is to prevent replay attacks. Signing messages with a digital signature is to prevent spoofing (who can view this message) and to ensure accountability (who sent this message). To sign a message, each WS-BA entity needs to have a certificate with a public/private key pair installed. We assume that a faulty entity has limited computing power to break the digital signature for each message.

3.2.3 The Lightweight BFT Framework

In this section, we described our lightweight BFT framework for achieving trustworthy coordination of Web Services Business Activities.

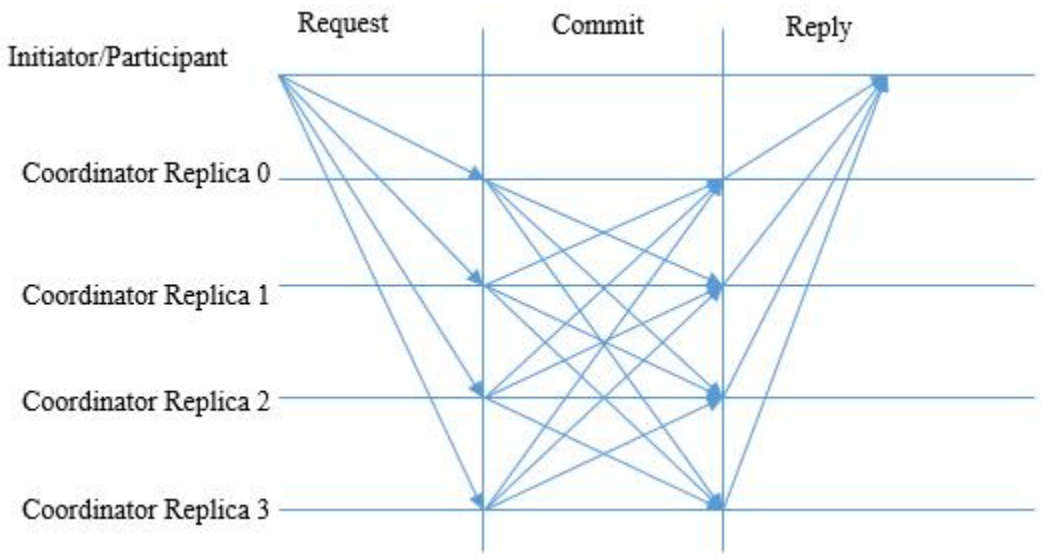


Figure 6 Normal operation of the lightweight BFT framework configured to tolerate one faulty replica.

The normal operation of the lightweight BFT framework is shown in Figure 6. The Initiator or a Participant broadcasts a client request <REQUEST, s, t, o, c>δ_c, to all Coordinator replicas. In such a request, s is the request sequence number, t is the timestamp, o is the operation to be invoked, c is the identifier of the client, and δ_c is the digital signature signed by the client c. Once a Coordinator replica received the request, it needs to validate the request by verifying its digital signature δ_c and checking if the sequence number s is for the request it expects.

If a client request sent from either the Initiator or a Participant is valid, the Coordinator replica sends a Commit request <COMMIT, c, s, t, d, k>δ_k, to all of the other Coordinator replicas. In the Commit request, c is the identifier of the client, s is the sequence number of the client request, t is the timestamp, d is the digest of the client request, k is the identifier of the replica, and δ_k is the digital signature of the Commit

request signed by the Coordinator replica k. A Coordinator replica may receive a Commit request before it receives the client request, it requests a retransmission of the client request by sending <SIMPLE-RETRANSMIT, c, s, d, k>δ_k to the Coordinator replica R that sent the Commit request. The Coordinator replica R retransmits the client request once it receives the retransmission request.

If a Coordinator replica can receive matching Commit requests (corresponding to the one sent by itself) from other 2f distinct Coordinator replicas, it processes the client request and update the coordination state if necessary. However, in the presence of Byzantine faulty Initiator, Participants and Coordinator replicas, it might happen that the replica fails to receiving 2f Commit requests matching the one sent by itself but might (1) receive 2f + 1 matching Commit requests but the digest in the Commit messages does not correspond to that of the client request the replica received or (2) fails to receiving 2f + 1 matching Commit requests like case (1) but receives at least 2f + 1 mismatched Commit messages from different Coordinator replicas.

For case (1), the replica just requests a simple retransmission of the client request with a digest that matches the 2f + 1 Commit requests. Once the replicas receives the correct client request, the replica abandons the original request and accepts the retransmitted one.

For case (2), the replica broadcasts a retransmit-with-proof request of the client request with proof of mismatched commitment to all other replicas and awaits until one of the following two events have occurred:

1) It has collected a retransmission with a valid Commit record with 2f + 1 signed Commit messages. This would enable the replica to commit and deliver the client request r in the new transmission right away.

Upon receiving such a retransmission-with-proof request, a replica should retransmits the request with matching request identifier (not necessarily the same digest d), and all the Commit messages collected for the same request. However, it could happen that the replica who received such a retransmit-with-proof request can be faulty and it could refuse or send retransmission of a Commit record to some replicas, so we define below mechanism to handle this case.

2) It doesn't receive a retransmission with a valid proof of commitment before a predefined timeout with value T has occurred. In this case, the timeout value is doubled to 2T and the replica broadcasts a new round of retransmit-with-proof request. This may be done recursively until condition (1) is eventually met. With reliable transmission guaranteed, as long as at least one non-faulty replica is able to commit a request r, all non-faulty replicas should eventually execute the request r with the digest by either committing it locally or receiving it from a retransmission with valid proof of commitment and no other requests conflicting with the request r should be executed at any non-faulty replica.

If the client (i.e., the Initiator or a Participant) that issued a request expects a reply from the Coordinator replicas, it collects matching replies from f + 1 distinct Coordinator replicas before it delivers the reply. The same mechanism is used for the Participants to handle (nested) requests sent by the Coordinator replicas. Because at most f Coordinator replicas are faulty and all of the replies match, at least one of the replies is sent by a non-faulty Coordinator replica.

If a reply is expected from the Coordinator replicas after they execute a client request, the Coordinator replicas send the reply <REPLY, s, t, r, k>δ_k to the Initiator or the Participant issued the request. In such a reply, s is the sequence number of the reply corresponding to the client request, t is the timestamp, r is the response result, k is the identifier of the replica, and δ_k is the signature of the reply signed by the replica k. Before delivers the reply, the Initiator or the Participant, which issued the client request, needs to collect matching replies from f + 1 distinct Coordinator replicas. Some of the client requests (e.g., the Complete/Close Participant requests from the Initiator in the steps (19) and (27) shown in Figure 4) might trigger the sending of nested requests (e.g., the Complete/Close requests in the steps (21) (22) (29) and (30) in Figure 4) by the Coordinator replicas to the Participants. The same mechanism as used for handling the client reply is used for the Participants to handle these nested requests.

In the WS-BA standard, the Activation service generates a unique Coordination identifier for each Business Activity. When the Coordinator is replicated, we should not have the Activation service in each replica generate the Coordination identifier independently and assign different Coordination identifiers to the same Business Activity. Even though this non-deterministic generation behavior could be controlled by using the mechanism in [40], we believe a more efficient method to handle this non-determinism by:

(1) Having the Initiator generate the Coordination identifier and piggyback it onto the Activation request so the Activation service in the Coordinator replicas can consistently assign the same Coordination identifier to the corresponding Business Activity.

(2) Having Coordinator replicas keep a history of the used Coordination identifiers and verifying the uniqueness of each newly proposed Coordination identifier against the history.

3.2.4 Proof of Correctness

In this section, we provide the proofs of correctness for the lightweight BFT algorithm based on the following properties:

P1. If a Byzantine faulty entity sends conflicting requests (different requests with the same request identifier) to different non-faulty Coordinator replicas, at most one of those requests is eventually delivered at all non-faulty Coordinator replicas.

P2. If a request is delivered at a non-faulty Coordinator replica, it will be eventually delivered at all non-faulty Coordinator replicas in the sender's source order.

P3. The states of the non-faulty Coordinator replicas will eventually converge, and the Initiator will eventually receive a response to a query regarding the state of the Business Activity, that is consistent with the converged state of the Coordinator replicas.

Proof of P1: Assume that there are two requests r and r' with the same sequence number sent from the same sender. The request r is delivered at a non-faulty Coordinator replica R while r' is delivered at another non-faulty Coordinator replica R'. It must be the case that a commit set S containing the proofs from 2f + 1 distinct Coordinator replicas which have accepted r and another commit set S' containing the proofs from 2f + 1 distinct Coordinator replicas which have accepted r'. Because there are 3f + 1 Coordinator replicas in total, at least f + 1 or more Coordinator replicas must be shared between S and S'. Also

there are only at most f faulty Coordinator replicas, so at least one Coordinator replica in the intersection must be non-faulty. However, a non-faulty Coordinator replica is not allowed to accept two different requests with the same sequence number from the same sender based on our BFT framework.

Proof of P2: If the request sender is not faulty, it establishes a reliable communication with each non-faulty Coordinator replica and reliably sends the request r in source order to that replica. If the sender is faulty, it might (1) only send the request r to some of the non-faulty Coordinator replicas, or (2) send conflicting requests to different non-faulty Coordinator replicas. Case (2) is not able to happen since the BFT framework guarantee P1.

In Case (1), if the request r is delivered at a non-faulty Coordinator replica R, the replica R must have received Commit messages for the request from 2f distinct Coordinator replicas matching the one R sent and at most f of them are faulty. If a non-faulty Coordinator replica R' did not receive the request, it will eventually receive the Commit messages for the request r from f + 1 distinct non-faulty Coordinator replicas, which will trigger a retransmission of the request. The replica R' will eventually receive the retransmitted request r from other non-faulty Coordinator replicas due to the assumption of reliable communication. Consequently, all non-faulty Coordinator replicas will eventually receive and deliver the same request r.

Proof of P3: Since the BFT framework guarantees P2, all non-faulty Coordinator replicas will eventually deliver the same set of requests and the states of all the non-faulty Coordinator replicas will eventually converge.

The BFT framework requires the Initiator to collect matched query responses from f + 1 distinct Coordinator replicas before accepting the response and at most f of the replicas might be faulty, therefore, the Initiator can be sure that at least one of the matched responses is sent by a non-faulty Coordinator replica. Consequently, the Initiator will eventually receive a response to a query regarding the state of the Business Activity, which is consistent with the converged state of the Coordinator replicas.

3.3 Implementation

In this section, we describe how we implemented the lightweight BFT framework and associated mechanisms mentioned in section 3.2.

We integrated our BFT algorithm and associated mechanisms with the Kandula framework [56]. Kandula framework belonged to Apache Web Services technologies and it has an open source, Java written, implementation of the WS-BA specification with the WS-BA-I extension. Also in this implementation package of the Kandula framework, we found a test application which exactly implemented the travel reservation example shown in Figure 4. The package contains necessary Apache Web Services libraries, including Apache Axis engine [57] and WSS4J [58] (an implementation of the WS-Security standard [59]) and logging utilities.

We implemented most of our mechanisms using Axis handlers as plugins. We can compose messages with additional information (e.g., the timestamp, sequence number, Coordinator identifier, security token, etc.) attached by different handlers created for

different purposes. We can easily test our framework implementation with different configurations just by adding or removing these plugins. E.g., we can test the system performance without security handling by removing our security plugin, an Axis handler implemented with WSS4J, from the processing pipeline. With the security plugin, all messages sent between different entities within a Business Activity are timestamped to prevent reply attack. We hash each of these messages to create its message digest and digitally sign the digest with the sender's private key to create the signature.

We replicated the Coordinator to tolerate f = 1 Byzantine faulty replicas and we totally need 3f + 1 = 4 replicas. The Initiator and the Participants are not replicated since they are acting as a client and it is not practical to replicate them. For the sake of benchmarking to have the experimental results are more consistent across different runs, we modified the test application (shown in Figure 4) with a slight modification. Instead of sending a reply to the Initiator as soon as sending a nested request ((19) and (27)) to all the Participants, a Coordinator replica waits the replies from all the Participants for the nested request until it has collected all the nested replies. E.g., a Coordinator replica is modified to send the Acknowledge reply (28) after it receives both nested replies (31) and (32).

We also implemented an adapted version of the PBFT algorithm [8] for comparing it with our lightweight BFT framework by modifying the existing Kandula framework. The PBFT algorithms are also implemented within Axis handlers which can be replaceable and easily customized. For fair comparison with our lightweight BFT framework, we enable concurrent total ordering of messages belonging to different Business Activities. The adapted PBFT implementation with the Coordinator replicated is configured to tolerate f = 1 faulty replicas.

3.4 Performance Evaluation

We carried our performance evaluation with different configurations both in a LAN testbed named MRILab and in a WAN testbed named PlanetLab.

The LAN testbed. The LAN testbed is configured as a Gigabit Ethernet, where the nodes are 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 OS. A 10KB message's transmission time would be less than 0.3ms.

The WAN testbed. All nodes in the WAN testbed are supported by international colleges from the whole world and they run the same software configured for the PlanetLab. The nodes we choose in PlanetLab have similar hardware configurations (e.g., Intel Core 2 Duo CPUs (2GHz to 2.6GHz)). They could be shared by multiple concurrent users so their actual load could vary from time to time.

We have the Coordinator replicas, the Initiator and the Participants each run on a distinct node. For each Business Activity, we measure its end-to-end latency at the Initiator and its throughput at the Coordinator. In each test run performed both in the LAN testbed and the WAN testbed, we obtained 1,000 samples to calculate median for the end-to-end latency and for the throughput. The reason that we chose to use the median instead of the mean is the results obtained in the WAN testbed may vary significantly across different runs due to unstable network load.

All results are obtained for four different configurations:

Unmodified Application. The test application is not modified for security protection.

Digital Signatures Only. The test application is only modified to have messages exchanged within all Business Activities are digitally signed so a faulty entity cannot disguise itself into some non-faulty entity to send messages. We use this configuration as the baseline for comparison.

Lightweight BFT. The test application is integrated with our lightweight BFT framework and associated mechanisms described in section 3.2.3.

Traditional BFT. The test application is integrated with an adapted PBFT (a traditional BFT algorithm) with the Coordinator replicated.

3.4.1 Performance Evaluation in a LAN

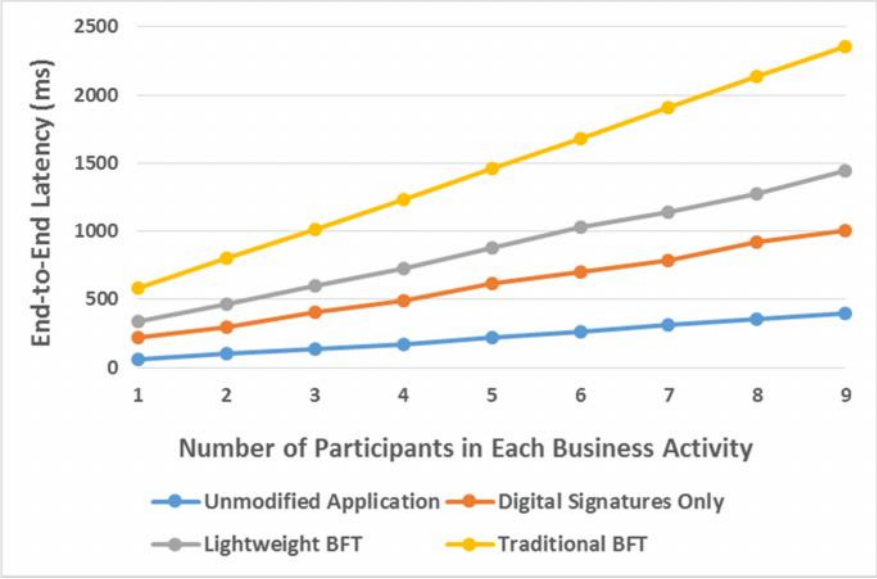
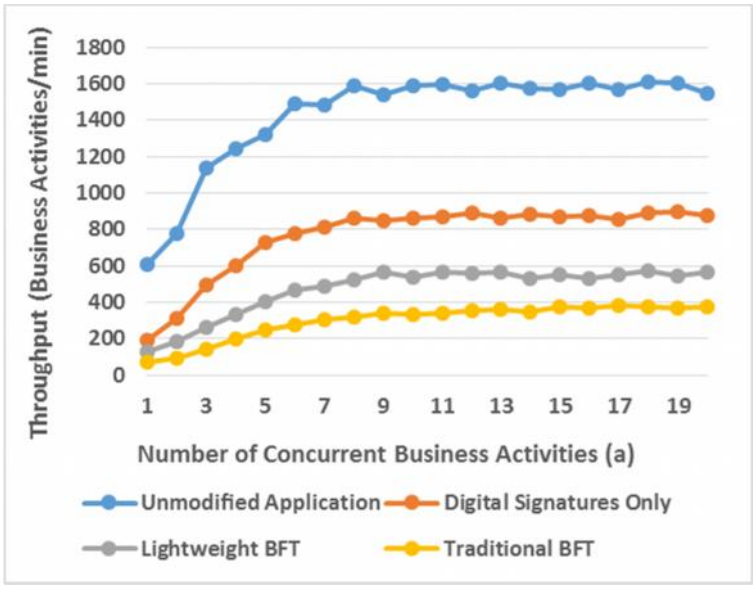


Figure 7 End-to-end latency in a LAN for Business Activities with different numbers of Participants under normal operation.

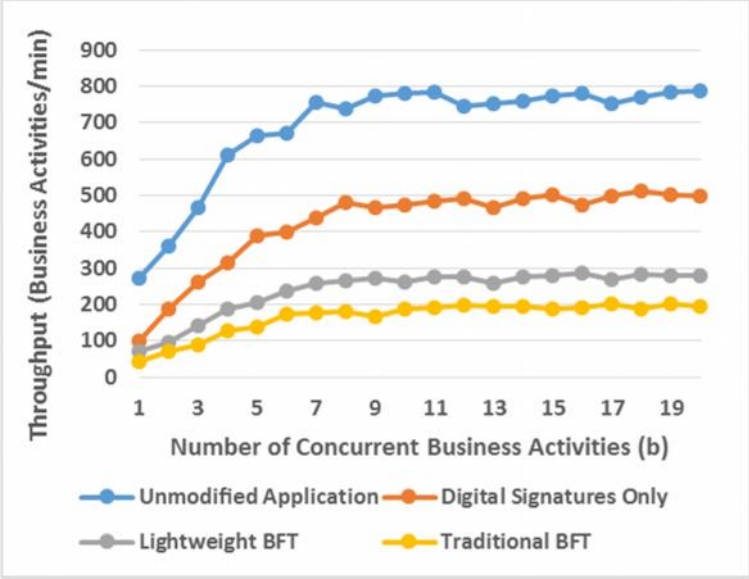
The end-to-end latency results obtained in the LAN testbed are shown in Figure 7. As can be seen in Figure 7, the end-to-end latency of the unmodified application in our LAN testbed is the smallest among all the four configurations, which is expected. The cost of signing a message in the LAN testbed is about 5ms and verifying a signed message is about 2ms. Due to the cost of signing and verifying every message exchanged within a Business Activity, the end-to-end latency of the test application with digital signatures is significantly larger than that of the unmodified one. As can be seen in Figure 4, a Business Activity with a single Participant involves 22 pairs of operations for message signing and signature validation on the critical path and 8 extra pairs of operations for message signing and signature validation for each additional Participant, which inevitably increases the end-to-end latency as the number of Participants increases. Due mainly to the use of digital signatures, our lightweight BFT framework also has much higher end-to-end latency than that of the unmodified test application, but we have to use the digital-signatures-only configuration as the baseline for comparison because it is essential to use digital signatures in real practice for non-repudiation in any serious Business Activity.

Besides the cost incurred by the configuration with digital signatures, the overhead of our lightweight BFT framework is also introduced by the commit messages during normal operation described in section 3.2.3. To commit a client request sent by the Initiator or a Participant, the normal operation (in Figure 6) includes one extra communication step, where one message signing for sending the commit request and 2f validation for the received commit requests from different replicas on the critical path. These result in about 40-50% larger end-to-end latency compared with that of the digital-signatures-only configuration.

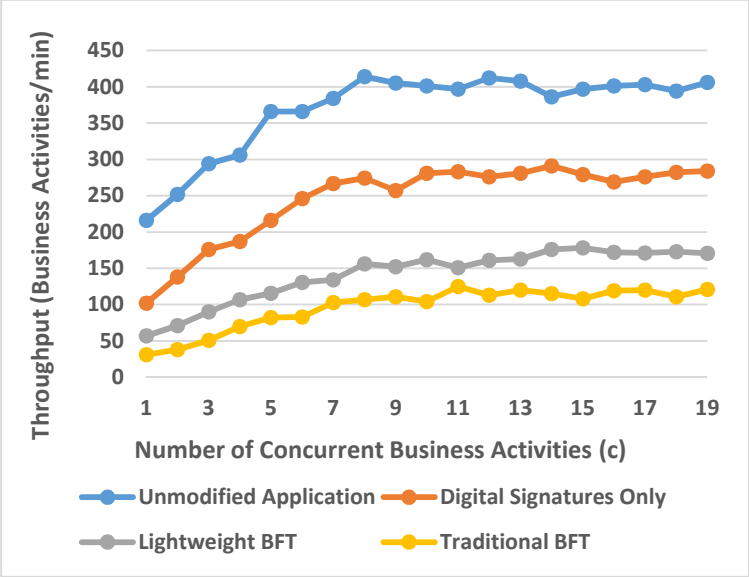
Compared to our lightweight BFT framework, the end-to-end latency with the adapted PBFT algorithm is about 130-170% higher than that of the digital-signatures-only configuration and almost as twice as that of our lightweight BFT framework (shown in Figure 7). The additional overhead is incurred by two extra communication steps (prepare phase and commit phase) for a client request agreement, where one message signing is for each phase, one validation for the client request and 2f received messages' validation for each phase on the critical path.



(a)



(b)



(c)

Figure 8 Throughput of the coordination services in a LAN with different numbers of concurrent Business Activities.

The throughput measurements are carried out for various numbers of concurrent Business Activities and the results are shown in Figure 8 (a) - (c). The experiments in our LAN testbed are performed for Business Activities involving with 2, 5, and 9 Participants with the same four configurations as those used for testing the end-to-end latency. The throughput reduction with the testing environment configured by our lightweight BFT framework is about 30% compared to that configured with only digital signatures in Figure 8 (a) - (c), which is expected due to including one extra communication step at the Coordinator. The throughput reduction with the configuration of the adapted PBFT is about 50%, much more than that of the lightweight BFT framework due to increased processing time caused by two extra communication steps at the Coordinator replicas.

3.4.2 Performance Evaluation in a WAN

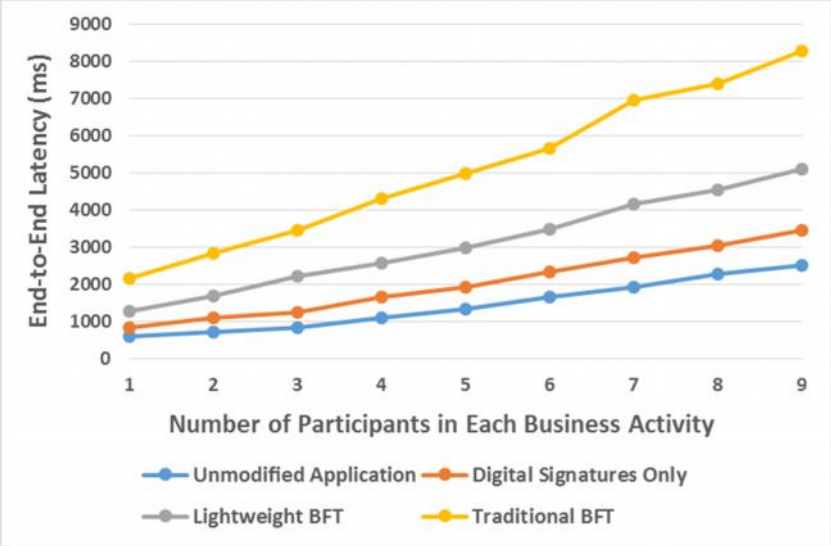
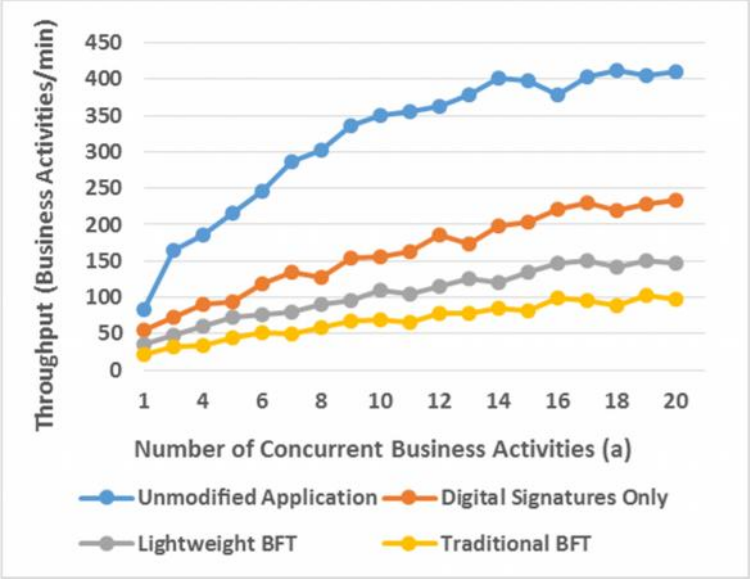
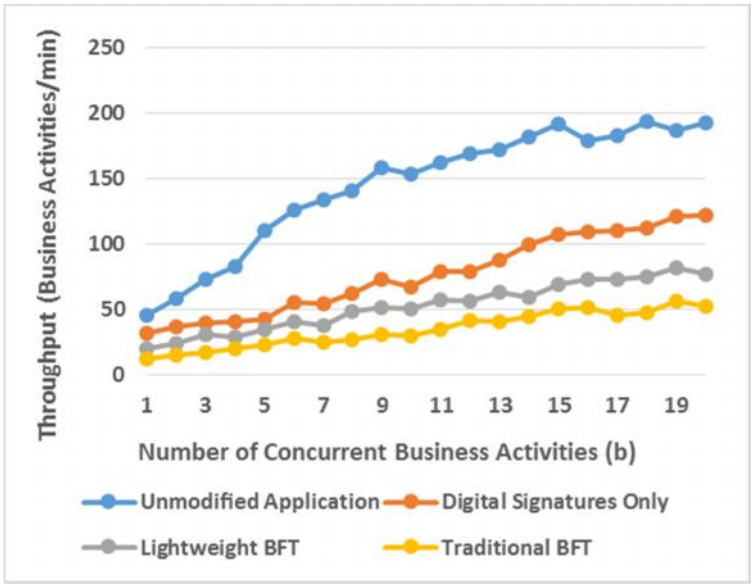


Figure 9 End-to-end latency in a WAN for Business Activities with different numbers of Participants under normal operation.

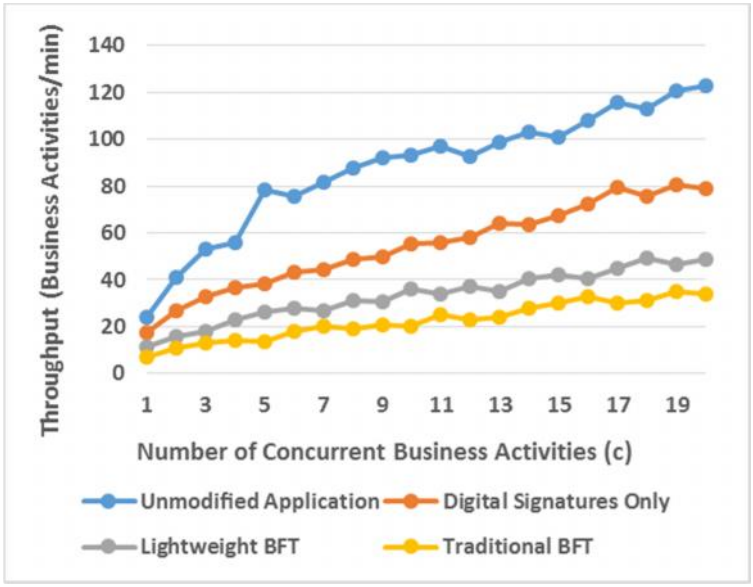
As can been seen in Figure 9, the end-to-end latency of baseline (digital-signatures-only configuration) is higher than that of the unmodified application obtained in a WAN testbed. This is expected since the cost of signing and verifying each message cannot ignored. However, because of the huge inevitable communication delay in the WAN environment, this relative cost of message signing and verification is reduced and as the consequence the end-to-end latency of our lightweight BFT framework drops from about 250% to about 100% compared with that of the unmodified application. And compared with the baseline, the end-to-end latency of our lightweight BFT framework is still higher but much less than that of the adapted PBFT.



(a)



(b)



(c)

Figure 10 Throughput of the coordination services in a WAN with different numbers of concurrent Business Activities.

The throughput measured in the WAN testbed (shown in Figure 10 (a) - (c)) is much less than that in the LAN testbed due to the inevitable larger message transmission time, which could be any of the following combined reasons:

Less capable CPUS. In general, the chosen nodes in PlanetLab are equipped with less capable CPUs compared to the chosen nodes in the LAN testbed.

Nodes are shared. In PlanetLab, the servers are shared by many concurrent users so multiple cores might not be available for all testing time.

Bandwidth limitation. In PlanetLab, the available network bandwidth between different servers is less than that in the LAN. Also the geographical distance and routing latency for PlanetLab servers can greatly decrease the effective bandwidth between different nodes. As a result, the message transmission time in the WAN testbed is much greater than that in the LAN testbed, which directly affects the throughput.

Probably because of the message transmission time is greatly increased and the relative cost of messages' signing and verification is reduced, the throughput reduction for our lightweight BFT framework is less apparent compared to that of the unmodified application, so is the adapted PBFT. However, the results obtained in the WAN testbed show that our lightweight BFT framework still outperforms the adapted PBFT implementation created from the traditional BFT algorithm.

CHAPTER IV

BFT FOR WEB SERVICES ATOMIC TRANSACTIONS

In this chapter, we use a banking application adapted from [56] (shown in Figure 11) as a running example, to illustrate the lightweight BFT framework we designed for protecting the integrity of the WS-AT Coordination services. The banking application is designed from the WS-AT specification and provides a banking Web Service with online transactions. The banking Web Service is a composite Web Service which consists of an Initiator, a Coordinator and several Participants, similar to the WS-BA specification. Each Participant has an Account service for maintaining its corresponding bank account.

A transaction process (illustrated in Figure 11) is started with customer's invoking the banking Web Service to transfer some money from one account to another. One round of 2PC is used to ensure the atomic commitment of a distributed transaction.

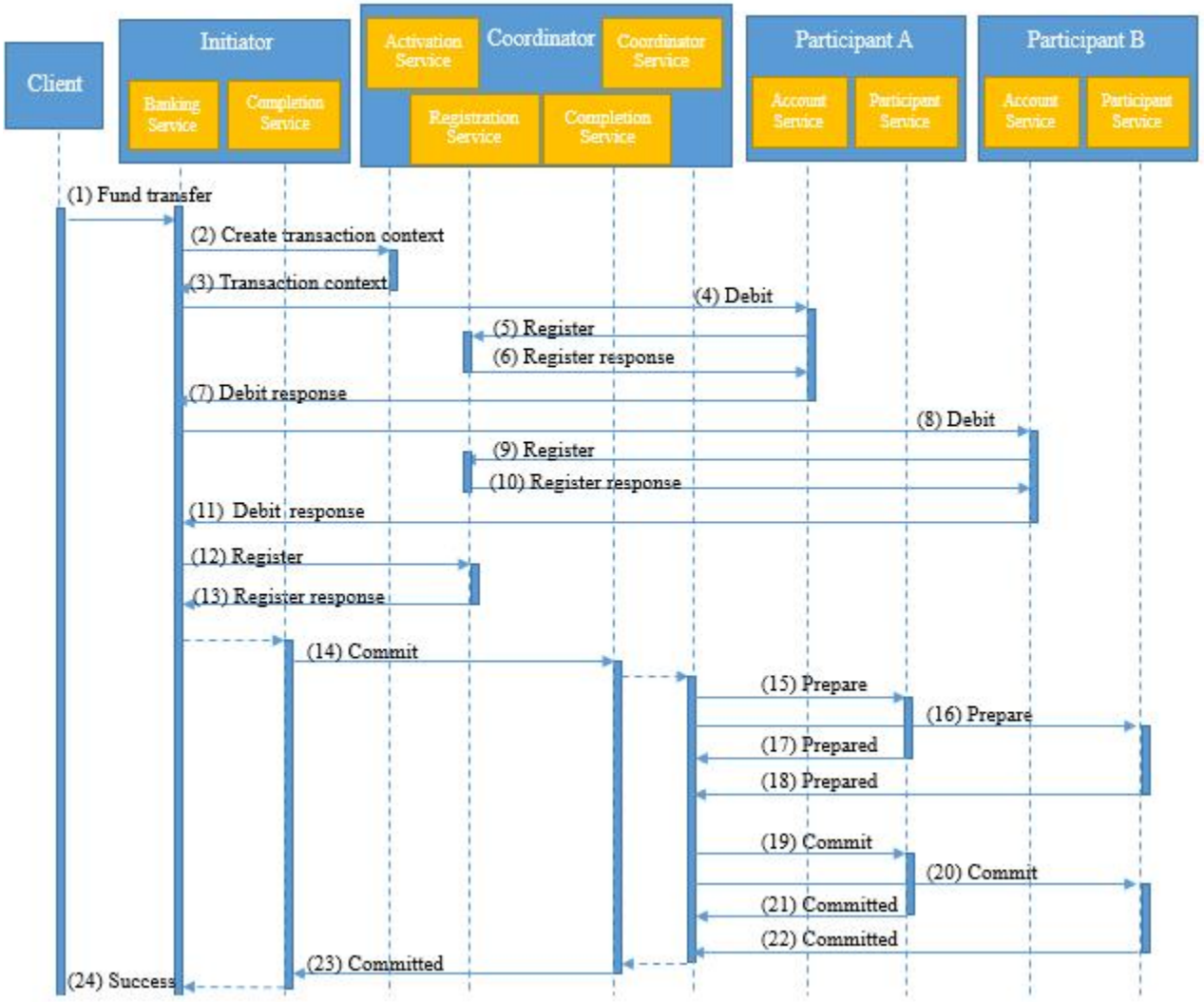


Figure 11 The sequence diagram for a banking application using the WS-AT protocol.

4.1 Threat Analysis

In this section, we analyze threats that can compromise the integrity of the WS-AT Coordination services. We do not consider general threats like service denial attacks since such threats are not in our research scope. In a WS-AT application like the example shown

in Figure 11, the threats could be from a faulty Coordinator, a faulty Participant and a faulty Initiator.

4.1.1 Threats from a Faulty Coordinator

Threats to the Activation Service. The Initiator requests the Activation service to generate a transaction context with a transaction identifier. A faulty Coordinator could use an invalid identifier for a new transaction. The identifier could be old, easy-to-predict or belonging to another transaction. These threats can be handled by the ways used for a WS-BA faulty Coordinator.

Threats to the Registration Service. A faulty Coordinator could register any unauthenticated entity with an illegitimate credential as a Participant. A faulty Coordinator could also register an authenticated Participant with an invalid endpoint reference. These threats cannot be easily mitigated using transport-level security mechanisms and we resolve them by our lightweight BFT framework described in section 4.2.3.

Threats to the Coordinator Service. To ensure atomic commitment of the distributed transaction, the Coordinator service runs the 2PC protocol. A faulty Coordinator could notify Participants to complete the transaction without the permission of the Initiator. Such a thread can be mitigated by requiring all notification messages to carry a security token which could be a signed certificate for the action from the Initiator. A faulty Coordinator could ask only some of the Participants to complete the money transfer and ask the rest to abort. Such threats can be handled by replicating the Coordinator and having voting of the received requests at the Participants. However, a fault Coordinator replica can

collude with a faulty Participant and have some non-faulty Coordinator replica commit the transaction and have the others abort. Such kind of threats would lead to inconsistent state among the Coordinator replicas and the Participants. We need to handle such threats by the lightweight BFT framework described in section 4.2.3.

Threats to the Completion Service. A faulty Coordinator could lie about the completion of the distributed transaction to the Initiator. We can replicate the Coordinator and have voting on the Initiator side to ensure only one decision can be accepted by the Initiator.

4.1.2 Threats from a Faulty Participant

Threats to the Coordinator. During the 2PC, a faulty Participant could either (1) lie about its execution status to the Coordinator or (2) send conflicting reports to different Coordinator replicas. E.g., for case (1), a faulty Participant could choose not to register with the Coordinator and not to participate in the 2PC. If a faulty Participant could involve in the 2PC, it could refuse to respond with a Prepared vote even if it tries to commit a transaction. It could also happen that the faulty Participant might vote to commit a transaction but abort the transaction locally. For case (2), a faulty Participant could respond a Coordinator replica with a Prepared vote and refuse to respond another Coordinator replica to have them make different decisions (one to Complete and another to Abort). To prevent threats for case (1), each Participant can be replicated and having voting of the messages at the Coordinator. However, replicating each Participant in real life is not practical. Therefore, we'd rather address this type of threat by signing each message sent between the Participants and the Coordinator and logging those messages at the

Coordinator. The threats for case (2) can be mitigated by using our lightweight BFT framework described in section 4.2.3.

Threats to the Initiator. A faulty Participant could attack the Initiator in a similar way as attacking the Coordinator. It could respond to the Initiator with a Debit response without performing the Debit operation on the account. It also could prevent the commitment of the distributed transaction by sending conflicting messages during 2PC to different Coordinator replicas. This type of threats can be addressed in a similar way that addressing similar threats for the Coordinator.

4.1.3 Threats from a Faulty Initiator

The integrity of a distributed transaction cannot be guaranteed if it is started by a faulty Initiator since the faulty Initiator could ignore responses from other entities, send arbitrary messages to other entities, or make arbitrary decisions during the entire lifecycle of the transaction. Such threats can be tolerated by replicating the Initiator.

A faulty Initiator may also send conflicting requests to distinct Coordinator replicas. Such a threat can be handled by the lightweight BFT framework described in section 4.2.3.

4.2 System Model and Solution Design

4.2.1 System Model Analysis

The Coordinator, the Initiator and the Participants run separately and they are coordinated by using a Coordinator object created for each transaction.

The Initiator is stateless. It starts and terminates distributed transactions and propagates a transaction to Participants with using a transaction context.

For each transaction, a distinct Coordinator object is created, which exists within the whole lifecycle of the transaction and provides services to coordinate entities involved in the transaction.

The Participants' states maintained by a Coordinator object are independent for distinct Participants and each Participant only can change its associative state. The change of the state associated with one Participant won't affect the state associated with a different Participant.

4.2.2 Assumptions

In the WS-AT specification, a distributed transaction is created to coordinate the interactions with other Web Services and help complete the transaction process. We assume such a distributed transaction does not involve nested transactions [60].

We assume the Initiator and the Coordinator are replicated by 3f + 1 Coordinator replicas to tolerate up to f faulty replicas during a transaction. The Initiator is stateless and

only 2f + 1 Initiator replicas are required to tolerate up to f faulty replicas. We assume that the clients and Participants can be Byzantine faulty. However, the Participants in a transaction are not replicated since in most real-world applications replicating Participants is not very realistic.

We assume that all messages exchanged between different service entities are protected by a security token which can be either a digital signature or a message authentication code. We assume that it is extreme hard to break the security token.

We assume each request has a request identifier and the corresponding response to the request attaches the request identifier.

We guarantee Byzantine agreement (BA) in our BFT framework on the key values (i.e., transaction id, registration set, and transaction outcome) needed for trustworthy WS-AT coordination. There exist a number of well-known general-purpose BFT algorithms [8] [11-13, 21, 61] that can be adapted to ensure BA. We assume using such an adapted algorithm requires 3f + 1 replicas to tolerate f faulty replicas. One of the replicas is selected as the primary and the rest replicas act as backups.

4.2.3 The Lightweight BFT Framework

In the presence of Byzantine faulty Participants, our BFT framework needs to ensure that all non-faulty Coordinator replicas and all non-faulty Participants agree on the same outcome of the transaction.

To protect the services and infrastructure of WS-AT coordination against Byzantine faults, our BFT framework comprises a suite of protocols and mechanisms described as below:

BFT Activation. The protocol ensures that all non-faulty Coordinator replicas choose and agree on the same transaction identifier for a new transaction, shown in Figure 12. Once it receives f + 1 matched Activation requests (with the same timestamp and client identifier) from different Initiator replicas, a Coordinator replica accepts the Activation request, and then multicasts a UUID Exchange message with a new generated UUID to all the Coordinator replicas. Once the primary can collect UUID Exchange messages from 2f + 1 distinct backups for the same transaction, it executes a Byzantine agreement (BA) on the UUIDs from the received messages. After the BA done, a transaction identifier is derived by performing a bit-wise exclusive OR operation on the agreed UUIDs to generate a combined UUID and all non-faulty replicas assign this transaction identifier to the Coordination object created for the new transaction. If an Initiator replica gets valid Activation responses from f + 1 distinct Coordinator replicas with the timestamp and client identifier matching those in the Activation request, the Initiator replica accepts the Activation response.

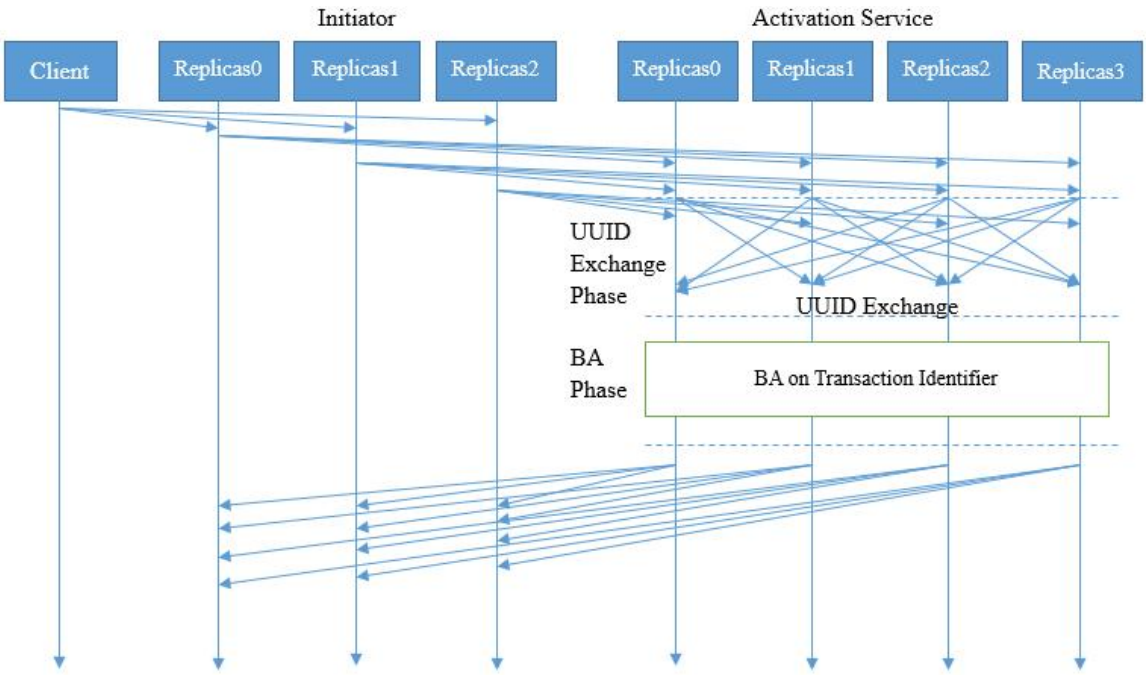


Figure 12 The BFT Activation service.

BFT Registration and Transaction Propagation. An Initiator replica sends an operation (e.g., fund transfer) request to a Participant and the Participant accepts the request only when it can collect at least f + 1 matching requests from distinct Initiator replicas. After it accepts an operation (e.g., fund transfer) request, a Participant registers itself by multicasting a Registration request to all Coordinator replicas, shown in Figure 13. A non-faulty Coordinator replica registers the Participant with the corresponding Coordinator object. The participant is considered registered successfully if it can collect 2f + 1 matching acknowledgments from distinct Coordinator replicas. All Participants need to respond the Initiator replicas with their registration status. If an Initiator replica notices that a Participant is not able to register to participate in the transaction process, it aborts the transaction. In this BFT solution, we perform the BA on the registered Participants but

defer it until we can combine it with the BA on the transaction outcome during the transaction commit phase. The registration for an Initiator replica is similar to that of the Participants.

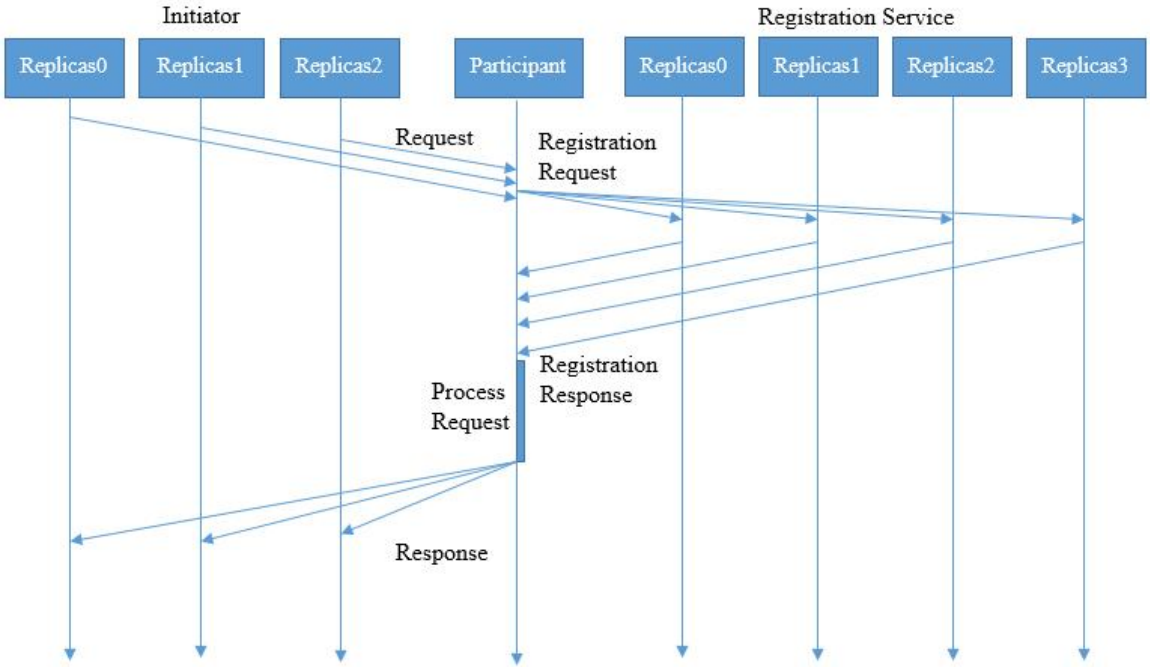


Figure 13 The BFT Registration and Transaction Propagation services.

BFT Completion and Distributed Commit. When an Initiator replica successfully has had all Participants complete all operations within a transaction, it requests all Coordinator replicas to help commit the transaction; otherwise, it requests them to rollback. Once it receives matching requests from f + 1 distinct Initiator replicas, a Coordinator replica starts the Commit or Rollback process. If a Coordinator replica needs to rollback the transaction, only a BA is needed to ensure that all non-faulty Coordinator replicas agree on aborting the transaction. If a Coordinator replica needs to commit the transaction, it starts the Registration Update phase by notifying all Coordinator replicas by sending them the records of all the registered Participants. When a Coordinator replica has collected

Registration Update messages from 2f other replicas for the same transaction, When a Coordinator replica has collected Registration Update messages from 2f other replicas, it checks to see if it has missed any registration records and add all missing 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 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.

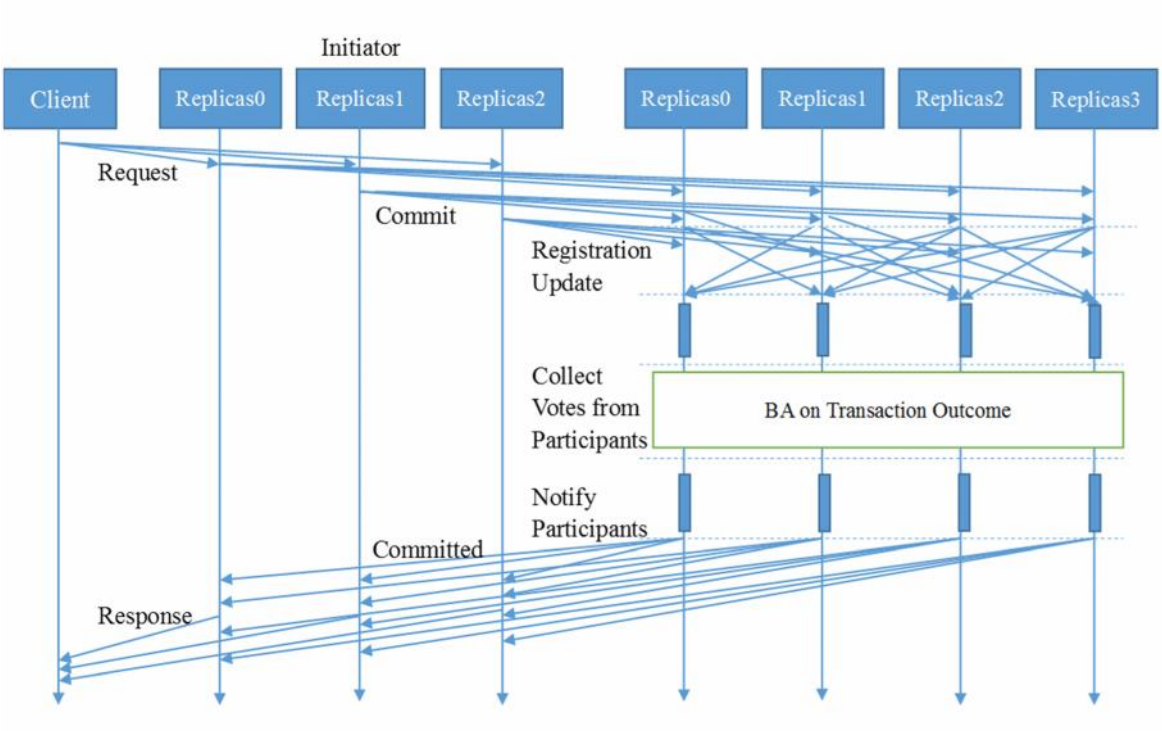


Figure 14 The BFT Completion and Distributed Commit services.

After it finishes the registration records exchange, a Coordinator replica starts the 2PC Prepare phase. A Participant defers accepting a Commit/Rollback request until it has collected matching requests from f + 1 distinct Coordinator replicas. At the end of the Prepare phase, a BA is performed to ensure that all non-faulty Coordinator replicas agree

on the same set of Participants and the same transaction outcome. At the end of the BA phase, a Coordinator replica runs the second 2PC phase to commit or abort the transaction and to notify the Participants the transaction outcome. A Participant accepts a transaction outcome when it has collected matching decision notifications from f + 1 distinct Coordinator replicas. After it ensures that all Participants have completed the transaction, a Coordinator replica responds the Initiator replicas with the transaction outcome. An Initiator replica accepts a transaction outcome if it has collected matching responses from f + 1 distinct Coordinator replicas.

4.2.4 Proof of Correctness

P1. All non-faulty Activation replicas generate the same transaction identifier and the transaction identifier generated is random.

P2. If a non-faulty Participant has successfully registered with the Registration service, then all non-faulty Coordinator replicas have a record of registration for that Participant.

P3a. If a transaction is committed at a non-faulty Participant, then it must happen that all non-faulty Participants have registered and sent Prepared.

P3b. All non-faulty Coordinator replicas should agree on the same outcome for the transaction.

P3c. All non-faulty Participants either commit or abort the transaction.

Proof of P1: Because the final UUID used to generate the transaction identifier is obtained deterministically from this set of UUIDs, the same transaction identifier is obtained by all non-faulty replicas. The Randomness is ensured by the generation of the UUIDs and the method used to combine the set of UUIDs with using the bit-wise exclusive-or operation.

Proof of P2: A non-faulty Participant needs to collect acknowledgments from 2f + 1 Coordinator replicas before it proceeds to the next phase. It means a set S1 of at least 2f + 1 Coordinator replicas have received the Registration request from a non-faulty Participant. During the Registration Update phase of the Completion and Distributed Commit protocol, a non-faulty Coordinator replica must collect the Registration Update messages from a set S2 of 2f + 1 Coordinator replicas (including itself). Because there are 3f + 1 Coordinator replicas, the two sets S1 and S2 must intersect in at least f + 1 replicas, among which at least one replica is non-faulty, which means that the registration records of this non-faulty replica must have been propagated to all non-faulty replicas.

Proof of P3a: Assume that a transaction is committed when with the existence of a non-faulty Participant that did not register or registered incompletely, did not vote, or voted Abort. We prove the property by contradiction.

i) If a non-faulty Participant failed before it could register or register completely, it would not have sent a reply to the Initiator, which would have led the Initiator to abort the transaction, contradicting the assumption.

ii) If a non-faulty Participant completed the registration, but failed before it could vote, all non-faulty Coordinator replicas would have timed out for the Participant, which would have led to the rollback of the transaction, again contradicting the assumption.

iii) If a non-faulty Participant voted Abort, a non-faulty Coordinator replica cannot commit the transaction since it is required to collect all Prepared vote from the registered Participants, again contradicting the assumption.

Proof of P3b: The property of ensuring all non-faulty Coordinator replicas agree on the same outcome for a transaction is ensured by the use of a BA algorithm on the outcome of transaction.

Proof of P3c: Assume that a transaction is committed at a non-faulty Participant p1 but is aborted at another non-faulty Participant p2. According to the Completion and Distributed Commit protocol, p1 committed the transaction if it received at least f + 1 matching Commit messages from distinct Coordinator replicas. Because at most f Coordinator replicas are faulty, p1 received a Commit message from at least one non-faulty Coordinator replica. By the proof of the first property, if any non-faulty Participant committed the transaction, all non-faulty Participants must have completed the registration and voted to commit the transaction.

However, because the non-faulty Participant p2 aborted the transaction, it must be one of the following cases:

(1) p2 unilaterally aborted the transaction due to it did not send a Prepared vote to the Coordinator replicas.

(2) p2 received a Prepare request, prepared the transaction, sent a Prepared vote to the Coordinator replicas, but received an Abort request from at least f + 1 distinct Coordinator replicas.

In Case (1), Participant p2 might or might not have finished the registration process. If it did not, an Initiator replica would have been notified of an exception, or would have timed out for p2. In either case, an Initiator replica should have decided to abort the transaction. If p2 had finished the registration process, there must exist a set S1 of at least 2f + 1 Coordinator replicas that have the registration record of p2. Because a BA algorithm is used on the decision of the Participants, it could not happen that the transaction is committed by the non-faulty Coordinator replicas with the inclusion of the Abort record in the decision certificate.

In Case (2), in terms of our BFT framework definition, because p1 has committed the transaction, at least one non-faulty Coordinator replica has agreed to commit the transaction. On the other hand, because p2 has received an Abort request, at least one non-faulty Coordinator replica has agreed to abort the transaction. It is contradicting the property that all non-faulty Coordinator replicas must have agreed to commit the transaction.

4.3 Implementation

We implemented our lightweight BFT framework and integrated our implementation with Kandula framework [56], an implementation of the WS-AT specification. We use a test application that Kandule provides, similar to the banking application example shown in Figure 11.

Since the initiator is stateless, it is replicated on 2f + 1 Byzantine faulty replicas to tolerate f = 1 faulty replicas. The Coordinator is replicated on 3f + 1 Byzantine faulty replicas to tolerate f = 1 faulty replicas. None of the participants are not replicated.

We implemented an adapted version of the PBFT algorithm [9] to demonstrate the benefits of using our lightweight BFT framework over the WS-AT application of using a traditional BFT algorithm. In the adapted version, an extra round of BA is used for each vote message from a Participant during the 2PC Prepare phase to ensure consistency among the Coordinator replicas in addition to a round of BA for the Activation and Registration requests.

We have all messages digitally signed by WSS4J HMAC [58]. We use this configuration as the baseline for performance evaluation.

4.4 Performance Evaluation

We carried our performance evaluation with different configurations both in a LAN testbed named MRILab and in a WAN testbed named PlanetLab.

The LAN testbed. The LAN testbed is configured as a Gigabit Ethernet, where the nodes are 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 OS.

The WAN testbed. All nodes in the WAN testbed are supported by international colleges from the whole world and they run the same software configured for the

PlanetLab. The nodes we choose in PlanetLab have similar hardware configurations (e.g., Intel Core 2 Duo CPUs (2GHz to 2.6GHz)). They could be shared by multiple concurrent users so their actual load could vary from time to time.

All results are obtained for four different configurations:

Unmodified Application. The test application is used as is, without any modification.

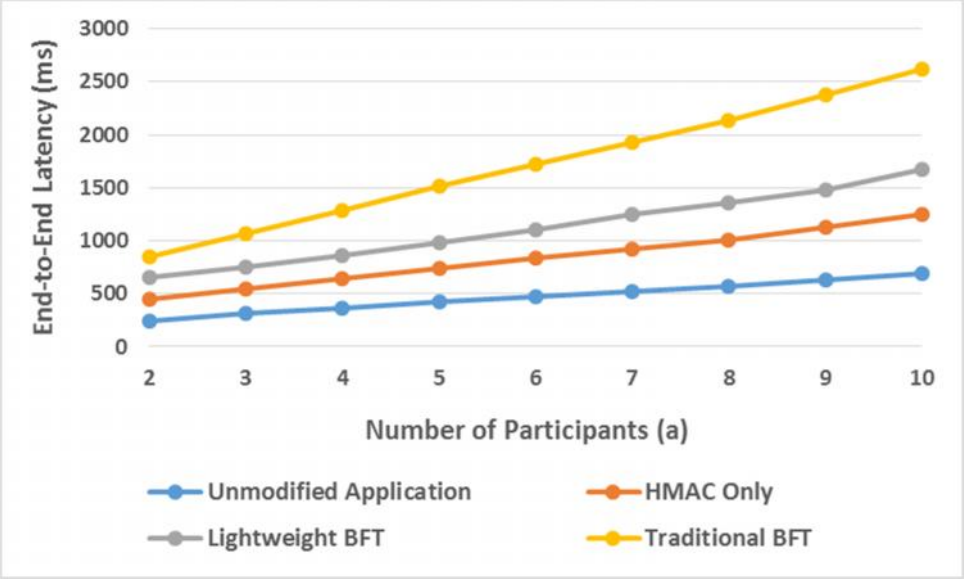
HMAC Only. The test application is modified so that all messages exchanged between different communication entities are protected by HMAC. We use this configuration as the baseline for comparison.

Lightweight BFT. The test application is protected by our BFT framework and associative mechanisms.

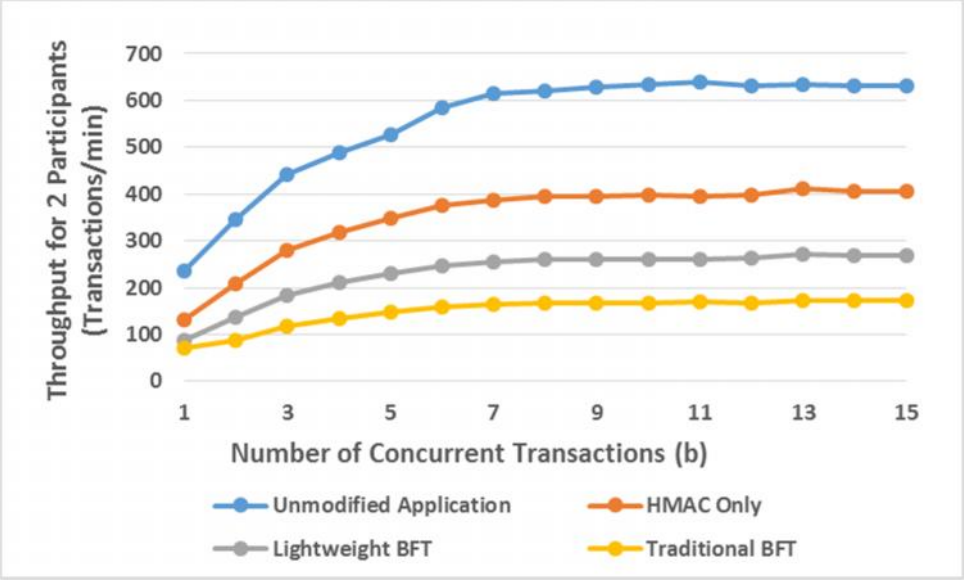
Traditional BFT. The test application is protected by an adapted PBFT algorithm implementation.

In all four configurations, the end-to-end latency is measured at the client side for the fund transfer operation shown in Figure 11. The throughput (the number of transactions per minute) of the distributed transaction is measured at an Initiator replica. We use various numbers of concurrent clients and Participants for the performance evaluation.

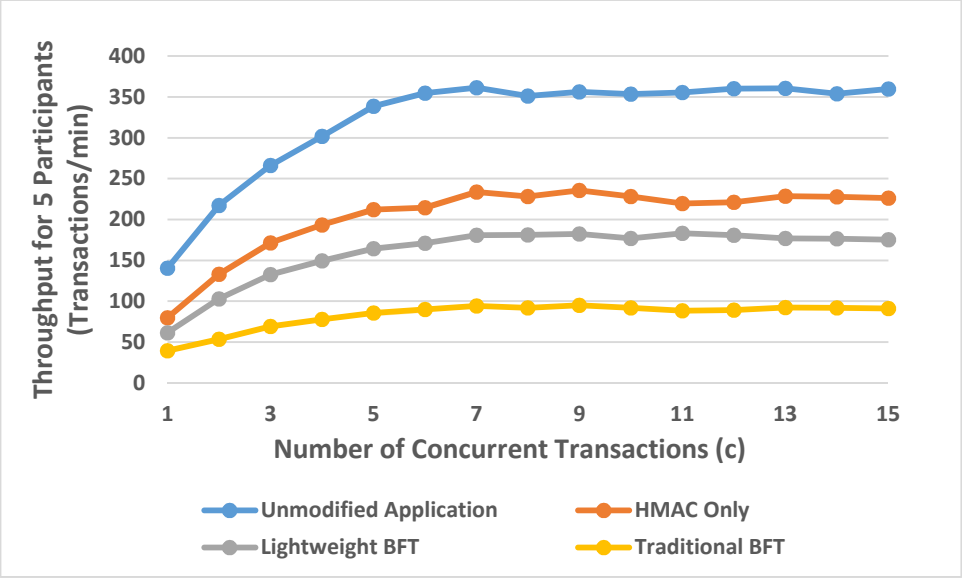
4.4.1 Performance Evaluation in a LAN



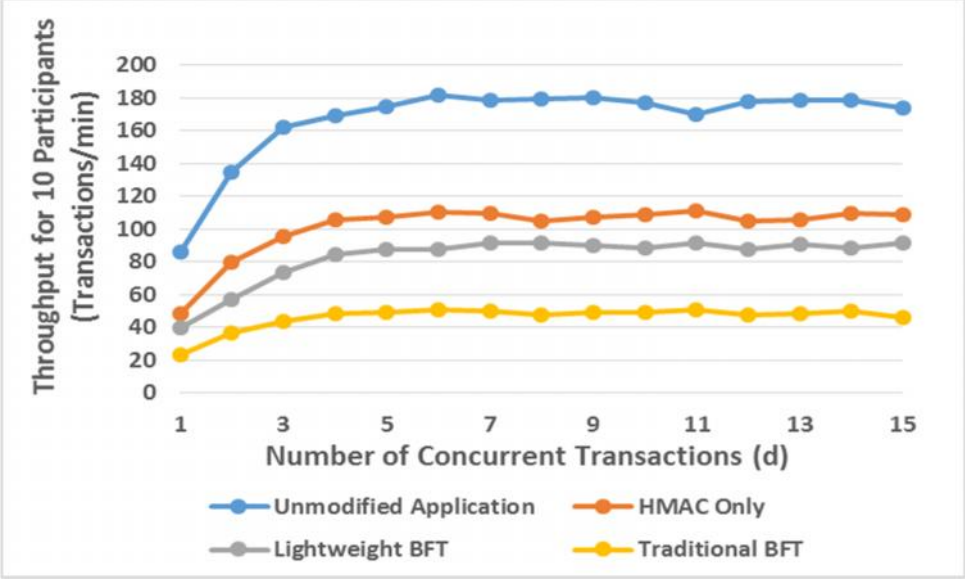
(a)



(b)



(c)



(d)

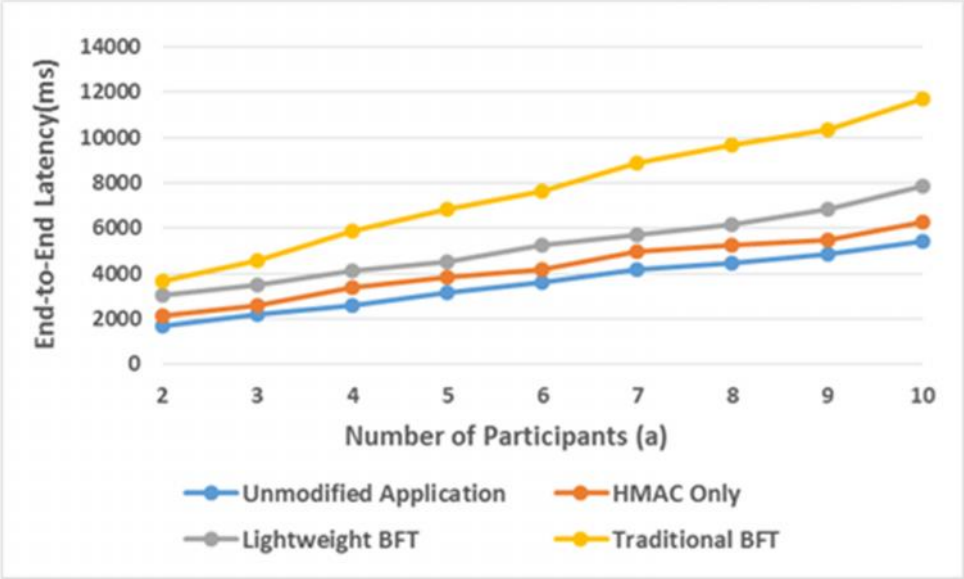
Figure 15 End-to-end latency and throughput of the test application in a LAN.

The end-to-end latency results in the LAN for f = 1 are shown in Figure 15(a). As can be seen from Figure 15(a), the end-to-end latency with our lightweight BFT framework

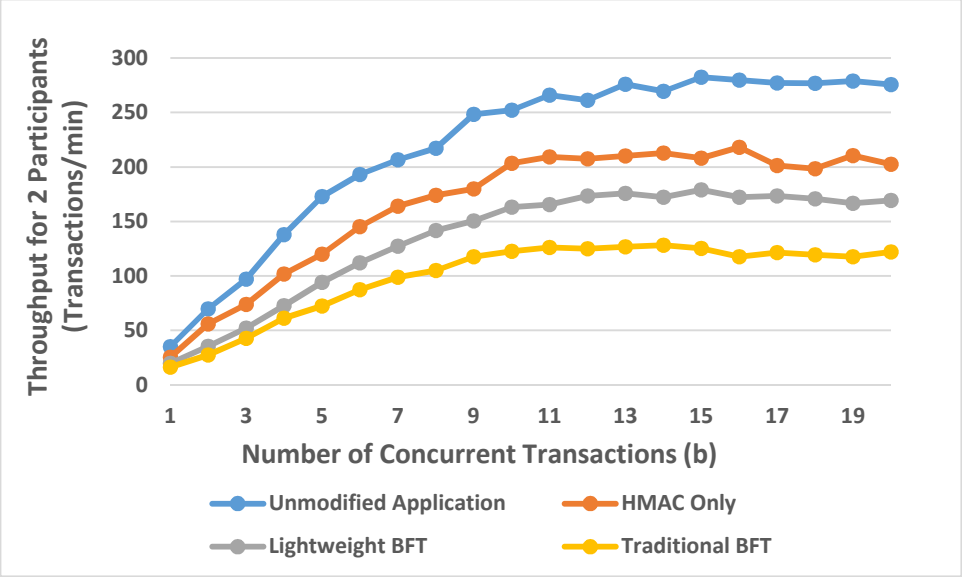
is significantly greater than that for the unmodified application but this increase is mainly due to the use of the HMAC to protect messages. We have to use the HMAC configuration as the baseline for comparison because it is essential for real world practice. Also from Figure 15(a), the end-to-end latency for the adapted PBFT implementation is much greater than that for our BFT framework, which is expected since more overhead is introduced from the adapted PBFT implementation than that is introduced from our BFT framework as more Participants involve in the transaction. Compared with the baseline, the relative overhead of our BFT ranges from about 45% with 2 Participants per transaction to about 25% with 10 Participants per transaction, and the relative overhead for the adapted PBFT implementation hovers around 100% for all the numbers of Participants.

The throughput results obtained in the LAN for f = 1 with 2, 5 and 10 Participants involved per transaction are shown in Figure 15(b)-(d). Compared with the baseline, the throughput reduction for our BFT framework ranges from about 45% with 2 Participants to about 15% with 10 Participants, which is expected due to the increased processing time at the Coordinator caused for messages' signing and verification. The throughput reduction, which ranges from about 50% to 60%, for the adapted PBFT implementation is much more significant than the throughput reduction for our BFT framework due to the increased processing time for messages' agreement.

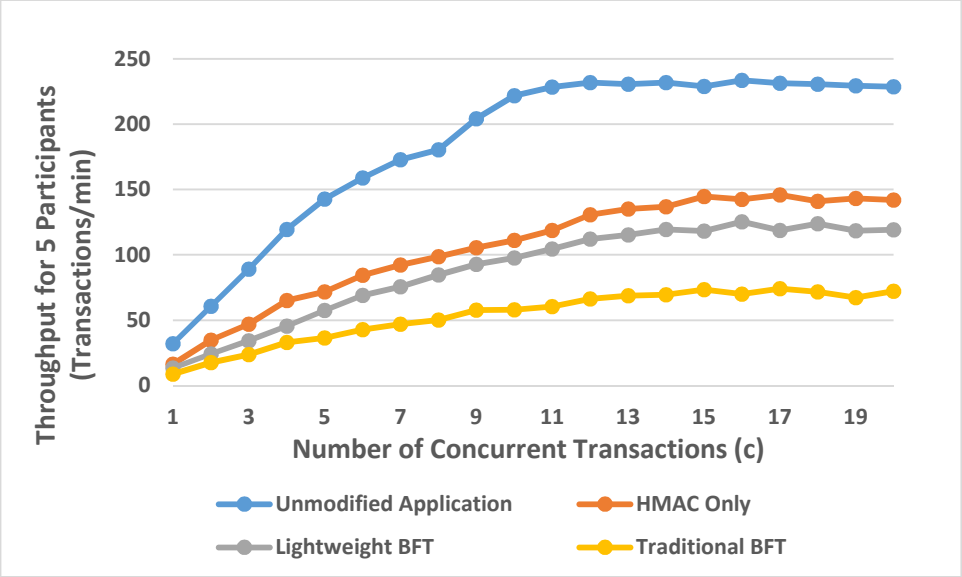
4.4.2 Performance Evaluation in a WAN



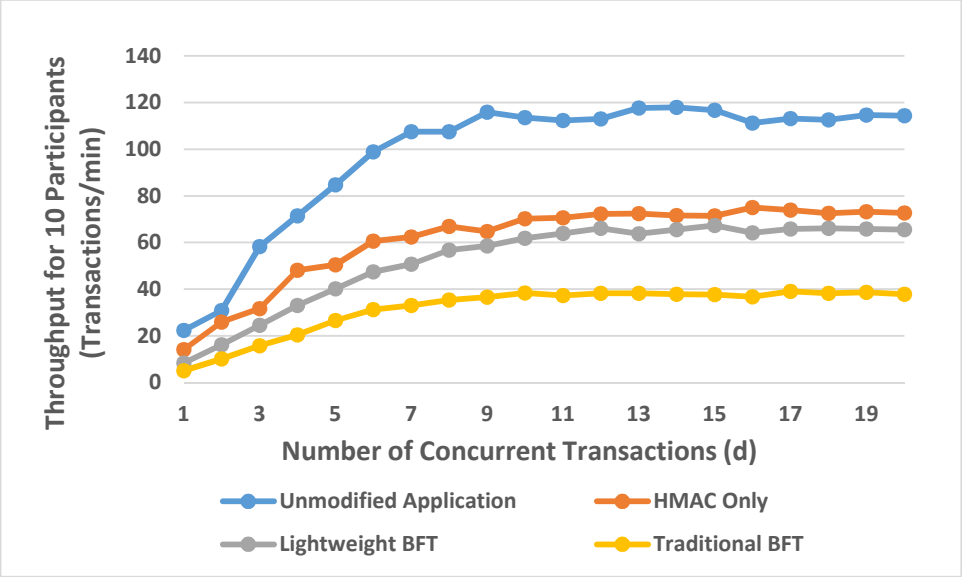
(a)



(b)



(c)



(d)

Figure 16 End-to-end latency and throughput of the test application in a WAN

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

Coordinator replicas. The average round-trip time between the primary and each Coordinator backup is approximately 40ms, measured by the “ping” command in Linux. We also run the Participants at PlanetLab nodes located at distinct locations. The round-trip time between a Coordinator replica and a Participant varies from about 20ms to 100ms.

The end-to-end latency results are shown in Figure 16 (a) and the throughput results are shown in Figure 16 (b)-(d). The results are obtained in the WAN for f = 1. The end-to-end latencies for our BFT framework and the adapted PBFT implementation have the same distribution trend. The throughput in the WAS for the baseline is reduced by about 25% to 45% compared with that in the LAN. For our BFT framework, the throughput reduction ranges from about 15% to 20%, and for the reference BFT implementation, the throughput reduction ranges from about 35% to 55%. The throughput reduction with respect to the baseline is more moderate for both our BFT framework and the adapted PBFT implementation. This is expected due to the much larger message transmission time for the WAN testbed than that for the LAN testbed since the network bandwidth in PlanetLab is significantly less than that in the LAN testbed and the CPUs in PlanetLab nodes are generally less powerful than those in the LAN testbed.

CHAPTER V

BFT FOR SESSION-ORIENTED MULTI-TIERED APPLICATIONS

In this chapter, a shopping cart application shown in Figure 17 is used as an example of session-oriented multi-tiered Web Services applications for threat analysis. Based on the analysis, we proposed a lightweight BFT solution for mitigating the threats which could potentially compromise the integrity of the middle-tier service. We also implemented a shopping cart prototype application based on the example shown in Figure 17 for performance evaluation. The experimental results show that the performance of our lightweight BFT solution is promising.

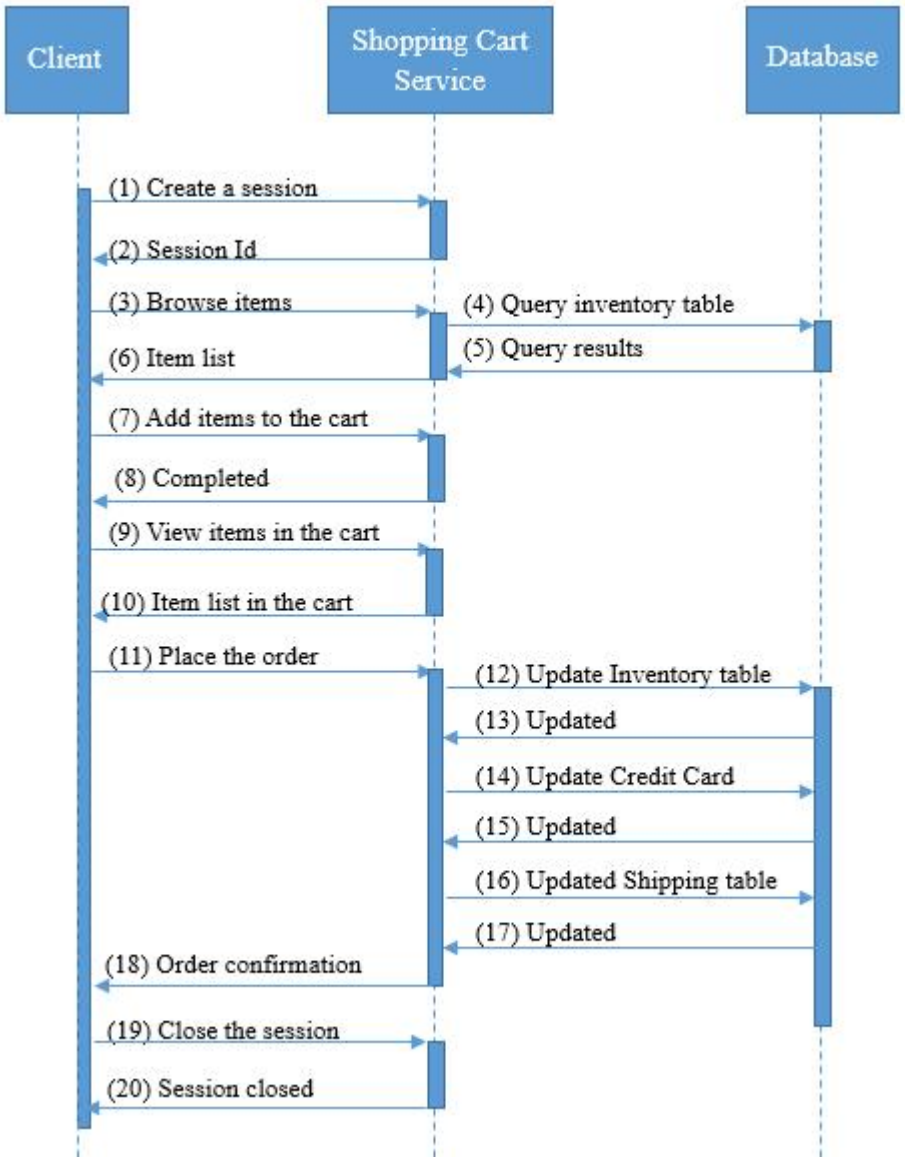


Figure 17 Web Service Shopping Cart Example.

5.1 Threads Analysis

To provide highly available and trustworthy services, it is essential for the middle-tier server to be protected against Byzantine faults considering the untrusted operating

environment of the Internet. In the rest parts of this section, we analyzed each type of threats which could be incurred by a faulty entity.

5.1.1 Threats from a Faulty Middle-Tier Server

A faulty middle-tier server could fail or refuse to respond to one or more client's requests either refuse to respond to one or more client's requests. For example, if the shopping cart service is not available to users, they may not be able to make any purchase, which could results in business loss. For the threats that the middle-tier service is unavailable to the client due a crash or malicious fault, we can mitigate such threats by replicating the middle-tier server, which is automatically handled by our lightweight BFT framework.

The integrity of the middle-tier service could be compromised such that it could lie about its execution status or alter the correct state saved at the backend database or insert spurious data into the database. If a faulty middle-tier server lies about its execution status, it could send replies that are inconsistent with the state stored at the backend database server to the clients. For example, a faulty middle-tier server for the shopping cart service shown in Figure 17 could also report to users that an unsuccessful order is successfully placed. A faulty middle-tier server could also alter correct data or insert spurious data into at the backend database server. For example, the middle-tier server could alter the purchased items by the user or add items into the order that the user did not purchase. The middle-tier server could charge the user's credit card without attaching the user's shipping address or attaching a wrong shipping address to the order at the backend database. If the integrity of the middle-tier service is compromised, in addition to replicating the middle-

tier server, a Byzantine Agreement (BA), which can be accomplished by any well-known BFT algorithm, on all incoming requests to the middle-tier server replicas to guarantee the relative ordering of these requests, can help eliminate such threats. Furthermore, the replies received at the client and the nested requests received at the backend database server must be voted on. We have designed a lightweight BFT solution to achieve the same aim and in this solution we demonstrate that total ordering is not necessary on the requests sent to the middle-tier server replicas. The lightweight BFT solution is discussed in section 5.2.

5.1.2 Threats from a Faulty Client

A faulty client could send conflicting requests to different replicas of the middle-tier server. For example, in the shopping cart application a faulty client could request to purchase different items at different replicas, request to add items to the shopping carts at some replicas and remove items from the shopping carts at other replicas, or request to only place an order at a part of the replicas. However, a faulty client could only corrupt the state associated with its own session and the state maintained for other session's clients will not be directly impacted. Therefore, we do not need to address such threats by replicating the middle-tier server and having BA on the requests sent to the middle-tier server replicas. It could happen that a faulty client could collude with a faulty middle-tier server replica to try to alter the correct data belonging to another client, we can mitigate such threats by having the requests received at the backend database server voted on if the faulty replicas are minority.

A faulty client could send a request with a used identifier to the middle-tier server. Such threats could cause replay attach but can be mitigated by having the client attach

timestamp to the request and have the middle-tier server replicas drop the requests with an old timestamp.

A faulty client could send malformed requests to the middle-tier server. Such threats cannot be addressed by any BFT algorithm since none of such algorithms ensure the validity of the requests. Like other BFT research, we only address how to ensure the integrity and consistency of the replicas.

5.2 System Model and Solution Design

5.2.1 System Model Analysis

States maintained in session-oriented multi-tier applications could be disjoint by different sessions (e.g., items in shopping carts are maintained by different sessions separately) or shared but only through the backend server (e.g., items listed in the inventory are shared among sessions). The backend server can ensure the execution order of requests sent to the middle-tier server replicas by synchronizing conflicting nested invocations triggered by executing the requests. The observation implies that:

(1) Requests belonging to different sessions can be executed in parallel and it is not necessary to do extra work to guarantee their relative ordering.

(2) Requests within the same session are delivered to different middle-tier server replicas in the same order because each session is involved with a single client and source ordering of such requests is adequate to ensure replica consistency. Therefore, it is not

necessary to ensure total ordering of the requests within the same session by using the Byzantine Agreement.

5.2.2 Assumptions and Requirements

In a session-oriented multi-tiered application, we made the following assumptions:

We assume that the middle-tier server is replicated by 2f + 1 replicas and at most f of them are faulty. None of the replicas plays the primary role since selecting one replica as the primary is not necessary for our BFT framework. We also assume that the backend database server can be trusted without using BFT replication.

We assume that all messages exchanged within each communication chancel are uniquely identified, which can be achieved by using system UUID generator or Web services facilities, such as Apache Axis2 [57].

We assume that the messages can be reliably exchanged between different entities in the application and an entity will eventually receive messages sent from others non-faulty entities. This reliability communication can be achieved either from transport level by using TCP or application level by using Web Services Reliable Messaging WS-RM [55].

We assume that all messages exchanged from different entities are all protected by a security token which could be either a digital signature or message authentication code. The security support can be satisfied by using techniques like WSS4J [58], an implementation of WS-Security.

5.2.3 The Lightweight BFT Framework

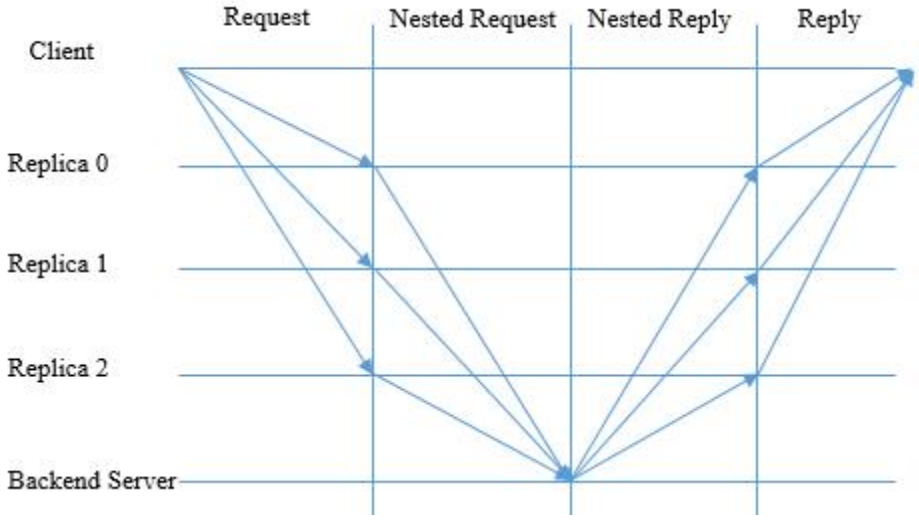


Figure 18 Normal operation of the lightweight BFT framework for session-oriented multi-tiered applications.

The normal operation (assuming f = 1) of our lightweight BFT algorithm for session-oriented multi-tiered applications is shown in Figure 18. A client broadcasts a request <REQUEST t, n, m, c>δ_c to all middle-tier server replicas. In such a client request, t is the timestamp, n is the request identifier, m is the message context (including the operation and all necessary parameters for executing the operation), c is the client identifier, and δ_c is the digital signature signed by the client c. The combination of t, n and c must be unique to each request. Once a middle-tier server replica received such a client request, it validates the request by verifying its digital signature δ_c and checking the validity of the request by comparing its logged history. The middle-tier server replica accepts and executes the client request if the request digital signature is valid and no other requests with

the same message identifier has been accepted. For simplicity, we use the request identifier of the first invocation of the client to determine the session identifier.

Executing a client request may trigger that the middle-tier server replica sends nested requests <DB-REQUEST s, t, n, sql, r>δ_r to the backend server. The nested request is a database request, where s is the session identifier, t is the timestamp, n is the sequence number of the nested invocation issued within the session s, sql is the SQL statement requesting to be executed at the backend server's database, r is the identifier of the middle-tier replica that sent the nested request, and δ_r is the digital signature of the replica. Once the backend server receives such a nested request R, it tries to collect at lease f + 1 nested requests matching R from distinct replicas. If the backend server can collect enough nested requests matching R, has not accepted a request with the same s and n before try to accept R, and verified R doesn't have an old timestamp, the backend server accepts the nested request R, executes sql and sends a nested reply <DB-REPLY s, n, r, sql-reply>δ_{db} containing the database response to all senders issuing the nested requests. In such a database server reply, s is the session identifier, n is the sequence number of the nested invocation, r is the identifier of the replica, sql-reply is the response of executing sql and δ_{db} is the digital signature of the backend server. A middle-tier server replica accepts such a nested reply only if the reply contains the same s and n corresponding to the nested request the replica sent. A middle-tier server replica may need one or more nested invocations. When the replica completes all the invocations and finish the execution of the client request, it replies to the client <REPLY t, n, rm, c, r>δ_r, where rm is the result

returned by the replica r. If the client c can collect at least f + 1 matching replies with the same n and c from different middle-tier replicas, it accepts the reply.

5.2.4 Proof of Correctness

Our lightweight BFT framework for guaranteeing the integrity of session-oriented middle-tier application satisfies the following:

P1. A faulty middle-tier server replica cannot make unauthorized changes at the backend server.

P2. Parallel execution of requests for different sessions cannot cause inconsistency among different non-faulty replicas.

P3. A non-faulty client will eventually receive a correct reply after it sends a request to the middle-tier server replicas and the change made by executing the request will be correctly effected at the backend server.

Proof of P1: The backend server cannot accept a nested request unless it has collected f + 1 consistent requests and there are at most f replicas are faulty, so it is obvious that they cannot effect any authorized changes at the backend server.

Proof of P2: If different non-faulty replicas of the middle-tier server want to access the persistent data at the same logic time, the nested requests issued by the replicas must carry identical session identifiers and the same sequence number. Furthermore, a replica must accept a nested reply that carries identical identifiers. This ensures that non-faulty replicas always have the same view of the persistent data.

Proof of P3: We first prove by contradiction on the claim that any state changes will be effected correctly at the backend server. If the state changes are not effected correctly at the backend server due to:

(1) A malformed request that is inconsistent with the client's intention or business logic, (2) the acceptance and processing of duplicate requests, or (3) the absence of a nested request that should have been accepted at the backend server.

For case (1), it implies that such a malformed nested request has been accepted at the backend server, which means at least f + 1 replicas have issued such a request. We assume a non-faulty replica does not issue malformed nested requests. Thus, this scenario is not possible because there are only up to f faulty replicas.

For case (2), each of the duplicate nested requests must have been accepted at the backend server, which means that at least f + 1 replicas have issued each of them and each of them carries a different message identifier. It is impossible that at least one non-faulty replica has issued duplicate message with different message identifiers.

For case (3), the absence of a nested request implies that no more than f replicas have issued the nested request, which is impossible because the request issued by a non-faulty client will reach all non-faulty replicas with reliable transmission and there are at least f + 1 non-faulty replicas.

The correct state changes at the backend server, together with correct processing at the f + 1 non-faulty replicas, ensure that the request is properly handled and a correct reply is sent to the client. Because there are f + 1 or more non-faulty replicas, it is guaranteed

that the client can receive at least f + 1 consistent replies, which ensures the delivery of the reply.

5.3 Implementation

Our lightweight BFT framework is implemented based on Apache Axis2 [57]. Most of the BFT components, such as the security support module (supported by the WSS4J [58] library), are implemented as pluggable Axis2 handlers which can be easily added into or removed from the framework without affecting the rest components. The backend database server is implemented by Apache Derby [62], an open source database system, with additional BFT voting mechanism integrated at the backend server. The simple shopping cart example shown in Figure 17 is implemented as our test application. We have all messages exchanged between different entities protected by HMAC (one of the security mechanisms supported by WSS4J).

5.4 Performance Evaluation

The performance of our lightweight BFT framework is carried out in a LAN testbed which consists of 14 HP BL460c blade servers connected by a Cisco 3020 Gigabit switch. Each blade server is equipped with two Xeon E5405 (2GHz) processors and 5GB memory, and runs a 64-bit version Ubuntu Linux OS.

The experimental data are collected for the following configurations:

Unmodified Application. The test application is not modified for security protection.

HMAC Only. The test application is modified to have all messages exchanged within the shopping cart service application protected with HMAC.

Lightweight BFT. The test application is protected with our lightweight BFT framework and associative mechanisms.

In all experiments for these configurations, we configure that at most one replica can be faulty. The end-to-end latency is measured at the client side that sends requests to the middle-tier server, which is the time between that the first client request starts a session and that the last response received and accepted to end the session. The throughput of the shopping cart service is measured at the middle-tier server (the shopping cart web service server), which is the number of sessions completed per second. We obtained around 1000 samples for each run and calculated the median for the end-to-end latency and for the throughput. The reason that we chose to use the median instead of the mean is that the median is more robust.

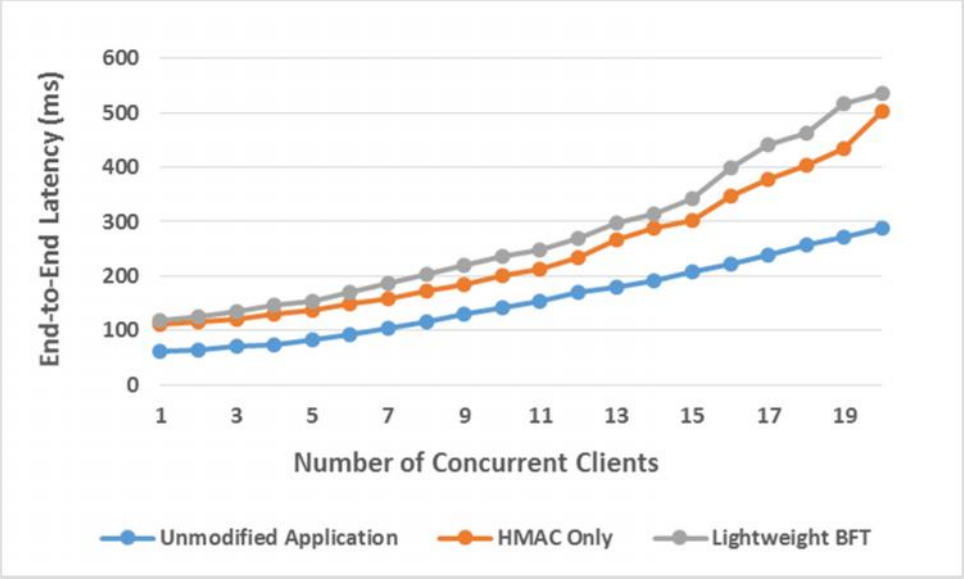


Figure 19 End-to-end latency with different number of concurrent clients.

As can be seen in Figure 19, the end-to-end latency of the HMAC only configuration is much larger than that of the unmodified application, which is expected since in the test application shown in Figure 17, each session has 20 steps and every message exchanged within the 20 steps is securely protected by HMAC. The cost of securing messages with HMAC varies by the message size. E.g., if the "browse" reply message in Figure 17 returns a list of 50 item objects, the cost of securing such a "browse" reply message takes about 6 ms and verifying this message takes about 12 ms. Signing a short message (such as "add an item to shopping cart") takes about 1 to 2 ms and verifying this message takes about 1 ms. The cost of securing all the messages inevitably increases the end-to-end latency. We use the HMAC only configuration as the baseline for comparison since it is essential to protect message communication in real practice. The end-to-end latency of our lightweight BFT framework is significantly larger than that of the unmodified application but this high latency is mainly due to message protection

supported by HMACs. The end-to-end latency of our lightweight BFT framework is less than 15% over the HMAC only configuration. The additional overhead incurred by our BFT framework is mainly due to the voting steps taking place at the client and the backend server.

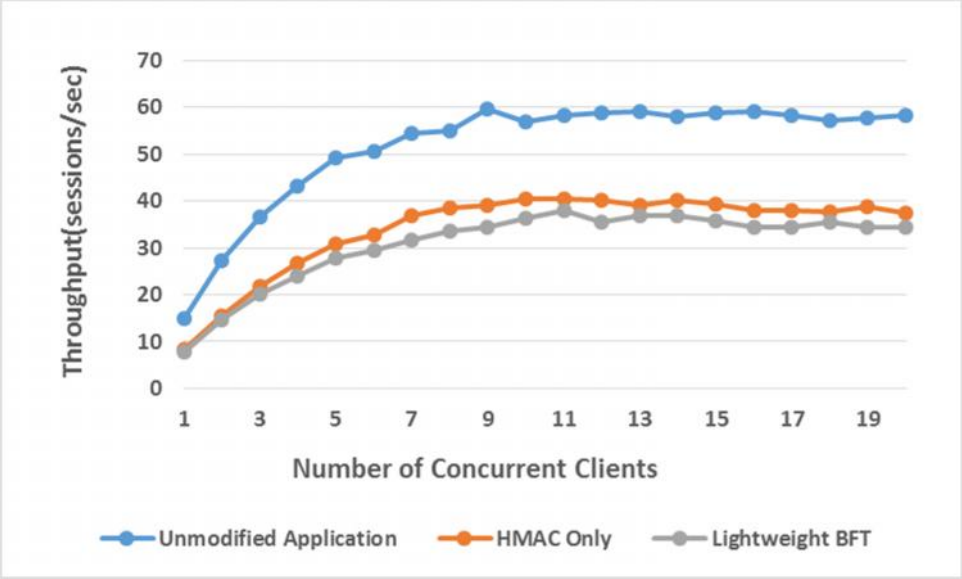


Figure 20 Throughput of the middle-tier server with different number of concurrent clients.

The throughput results are shown in Figure 20, which show that the throughput reduction with the baseline and our lightweight BFT algorithm is dramatic compared to the unmodified application. The overhead is mainly caused by the message securing supported by using HMACs. However, the average throughput reduction with our lightweight BFT algorithm is less than 15% compared to the HMAC only configuration (the baseline).

In this experiments, we do not have a configuration with a traditional BFT solution for comparing with our BFT solution, since based on performance evaluation carried for the BFT solutions we designed for WS-BA and WS-AT applications, supporting total

ordering for all client requests and nested backend server replies to middle tier server replicas will inevitably incur significant runtime overhead. For each client request in a session, it only could happen:

(1) A client request is executed by middle tier server replicas according to its source order and no nested invocation is required by this client request.

(2) A client request is executed by middle tier server replicas according to its source order but it requires nested invocation at the backend server.

If total ordering is supported for case (1), before the client sends the next request, it needs to wait for the current request get totally ordered first and then executed by the middle tier server replicas.

If total ordering is supported for case (2), the client also needs to wait for the current requests get totally ordered first. However, due to nested invocation, the nested reply from the backend server also needs to be totally ordered before the server replicas are able to execute the client request.

For the application example shown in Figure 17, it requires 4 total ordering (for case (1) client requests) and 6 total ordering (for case (2) client requests). For one round of total ordering in the PBFT algorithm, the additional overhead is incurred by two extra communication phases for a client request agreement, where one message signing is for each phase, one validation for the client request and 2f received messages' validation for each phase. Based on the analytical data, signing a message takes about 2 ms and verifying this message takes about 1 ms. Using such analytical data, we can predict that the cost of one round of PBFT should take about 9ms and the end-to-end latency for a client to

complete a session is about 90ms. Since the overhead incurred by the total ordering supported by a traditional BFT algorithm will be very obvious based on the analysis, we do not need to configure a traditional BFT algorithm to compare the performance with our BFT solutions.

CHAPTER VI

BFT FOR EVENT STREAM PROCESSING

In this chapter, we present a lightweight BFT framework with associative mechanisms that help achieve BFT on stateful event stream processing of autonomic computing systems. The BFT mechanisms are designed for helping event consumers make consistent decisions based on processing outcomes. In our lightweight BFT framework, we only consider stateful event processing because autonomic computing systems typically do not involve stateless event processing.

6.1 Threat Analysis

In this section, we analyzed potential threats that could compromise the integrity of event stream processing of autonomic computing systems. We do not consider general threats like service denial attacks since such threats are not in our research scope.

6.1.1 Threats from a Faulty Event Producer

A compromised event producer could try to lead to absence or wrong decisions of the autonomic computing systems by omitting events or emitting faulty events to affect the decisions of the event streaming processing. These threats can be addressed by replicating the EPA and having all non-faulty replicas make their decisions based on the same set of events. However, the set of events may not contain events affecting the decisions or contain faulty events, which could cause a validity issue that all non-faulty replicas would make the same wrong decisions. This issue may need to be addressed by approaches as below:

Incorporating redundant event producers. With involving redundant event producers to help cross-examine events, the omitted or faulty events sent can be masked and excluded.

Using strong pattern matching. Strong pattern matching or derivation strategies can help detect events which are omitted or faulty. For example, deriving results by computing the median value for a set of events' properties is more robust than deriving results by computing the average value.

Even if the EPA is replicated and having mechanisms to detect the presence of faulty events and the absence of events affecting the decisions, a faulty event producer could send conflicting events to different EPA replicas to cause them reach inconsistent decisions. In this case, one round of state synchronization among all non-faulty EPA replicas is necessary in order to ensure all the non-faulty replicas compute the decision on the same set of events.

6.1.2 Threats from a Faulty EPA

A faulty EPA could refuse to send decisions, fake decisions or send decisions inconsistent with the ones it really made. Such types of threats can be mitigated by replicating the EPA and using voting on the decisions from different EPA replicas at each event consumer.

A faulty EPA replica could collude with a faulty event producer and send conflicting decisions to event consumers. This is a consistency issue which could be solved by using BA on the decisions sent by the EPA replicas.

6.2 System Model and Solution Design

6.2.1 System Model Analysis

The event stream processing system consists of a list of event producers, one EPA, and one or more event consumers, as shown in Figure 3. The event producers issue events to the EPA and each event is timestamped. The EPA may partition the received events based on the context identifier attached for each event. A context partition may change all the time during runtime events processing. e.g., the EPA could use a time-based processing window with limited size to maintain and arrange events by their timestamps. An out of date event will never be added to the processing window and the oldest event in the event will be eliminated from the window when new event arrives and the size of the window is already reached.

The EPA processes the events in context partition based on a set of predefined rules. In this research, the EPA is configured aggregation oriented, which computes one or more attributes of a sequence of events (e.g., all current events in a time-based window) to derive higher level events. During the derivation step, a new event (or a decision) could be derived by translated, enriched, or projected and consumed by event consumers which registered to listen to the decision.

6.2.1 Assumptions and Requirements

In an event stream processing application, we made the following assumptions:

We assume that the EPA is replicated by 3f + 1 replicas and at most f of them are faulty. None of the replicas plays the primary role since selecting one replica as the primary is not necessary for our BFT framework.

We assume the BA cluster is composed of 3f + 1 ordering nodes and all the nodes run the same BFT algorithm (which could an adapted version of any traditional BFT algorithm). One of the replicas acts as the primary which leads the total ordering of incoming messages to the BA cluster. A BA operation can ensure that all non-faulty EPA replicas receive the same set of messages in the same total order.

We assume that all messages exchanged between different entities are:

- Reliability guaranteed by their communication channel like TCP.
- Protected by a security token like a digital signature or message authentication code.

We assume events arrived in each context partition are timestamped and ordered in the partition determined by properties like the timestamp.

6.2.3 The Lightweight BFT Framework

Our BFT framework includes two different modes: normal operation and state synchronization modes.

1) Normal Operation: An event producer issues an event to all EPA replicas and the event is timestamped and contains a context identifier. Once an EPA replica received such an event, it puts the events to its context partition by checking context identifier, processes the context partition based on predefined rules and notifies the registered consumers the decision if any derived. A decision message produced by EPA replicas has the form <DECISION, i, s, h, c, dc>δ_i, where i is the identifier of the replica, s is context identifier, h is the history hash value, c is the identifier of the event consumer listening to the decision, dc is the content of the decision and δ_i is the signature signed by replica i.

The history hash h is a hash computed for the history of a list of events e_1, e_2, ..., e_n in a context partition. The format of the hash is $h = hash(e_1) \oplus hash(e_2) \oplus \ldots \oplus hash(e_n)$, where hash() can be any general hash function like SHA-1, and $\oplus$ is the exclusive OR operator. The value of the history hash is independent from the relative ordering of the events, e.g., $h(e_1, e_2) = h(e_2, e_1)$.

An event consumer needs to register to listen to the decision sent from the EPA. After a consumer registered what decision it wants to be notified, it waits to receive 2f + 1

messages matching for the same decision from different EPA replicas. If the decision messages have the same context identifier s, the same history hash h, and the same decision content dc.

The event stream system operates in the normal operation mode as long as no faulty event producer presents and there are only up to f faulty EPA replicas in the system.

2) State Synchronization: An event consumer might not be able to collect at least 2f + 1 matching decision messages if faulty event producers send conflicting events to different EPA replicas. The event consumer starts sets a timer until it has collected 2f + 1 mismatched decisions and waits for the rest and if a timer expires, it requests a round of on-demand state synchronization (shown in Figure 21) by sending a decision mismatch message <DECISION_FAIL, c, s, DS>δ_c to all EPA replicas. In the sent message, c is the identifier of the event consumer, s is the context identifier included in the decision messages, DS is the set of received, digitally signed decision messages (as the proof for the need of state synchronization), and δ_c is the signature signed by the consumer c.

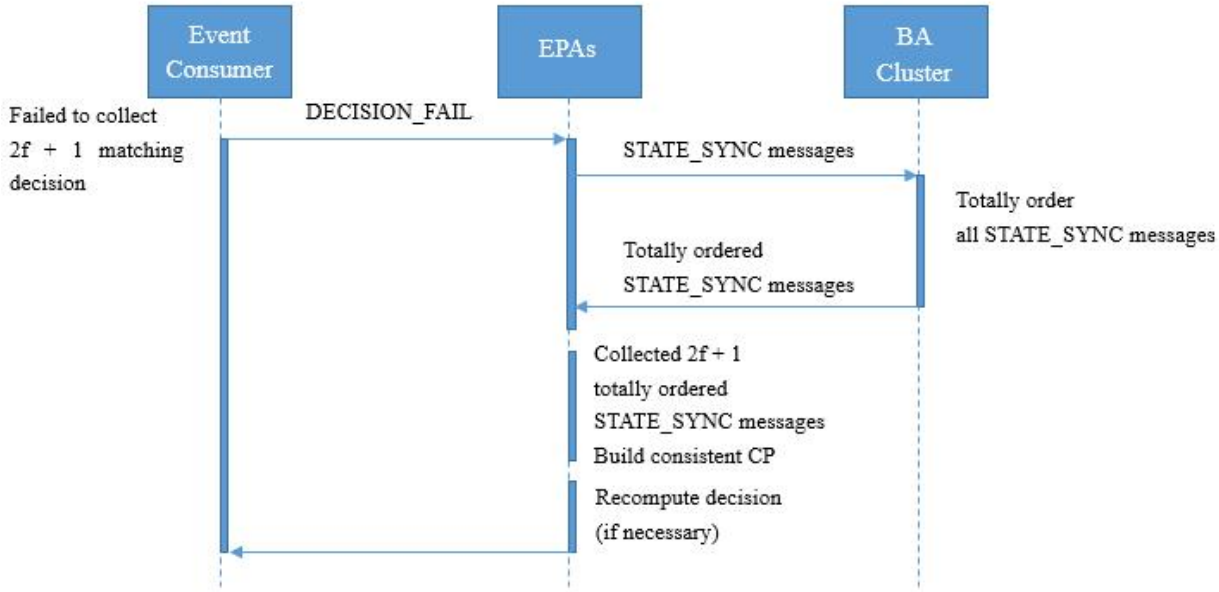


Figure 21 On-demand state synchronization.

Once receiving a message notifying decision fails to being reached by an event consumer, a non-faulty EPA replica validate the message by checking the set of decision messages DS included in the DECISION_FAIL message. If the DS is invalid, the event consumer is apparently compromised and should not be notified by any decision or have its message accepted by any EPA replica in the future; otherwise, the EPA sends a state sync request <STATE_SYNC, i, s, CP_s^i >δ_c to the BA cluster for grouping and total ordering. The state-sync message sent by replica i has the form where s is the context partition identifier, CP_s^i is the list of event records in the partition at replica i. Each event record consists of an event identifier and the digest of the event. A faulty event consumer could attempt to trigger unnecessary rounds of state synchronization, which is an expensive step. This type of threats can be controlled by using a non-forgeable proof for each state synchronization request.

When the EPA is replicated, not only a faulty event producer but a faulty EPA replica could disseminate conflicting events to other replicas during a round of state synchronization. The threat can be addressed by supporting BA using a general-purpose BFT algorithm such as PBFT. In this operation mode, we perform BA on the received state-sync messages from different EPA replicas to ensure all these messages for the same context partition is totally ordered. Upon receiving the first 2f + 1 totally ordered state-sync messages sent by different replicas, an EPA replica proceeds to building a consistent context partition CP(s) according to the following rules:

(1) An event is included in CP(s) if there are state-sync messages including the same event (with the same timestamp, context identifier and event content) and no events conflicting with this one are found among the received messages.

(2) If conflicting events (with same context identifier but different event content) are detected in the state-sync messages, at most one of the events can be included in CP(s) due to rule (1). The faulty event producer sent the conflicting events will be blacklisted and rejected to accept any future event.

(3) An event included in less than f + 1 state-sync messages with events conflicting with this one detected is not included in CP(s).

Once CP(s) is built up after elaborating all the first 2f + 1 received state-sync messages, an EPA replica compares CP(s) with its own context partition. If they are not identical, the EPA replica replaces its own context partition with the built CP(s), and repeats the processing and derivation step. It is guaranteed that all non-faulty EPA replicas are consistently generating the same decision using CP(s) so that a registered non-faulty event consumer can accept the decision.

6.2.4 Proof of Correctness

In this section, we described our BFT mechanisms for protecting the integrity of event stream processing. Our BFT mechanisms can satisfies the following properties.

P1. If a decision is accepted, it must be generated by the majority of non-faulty EPA replicas.

P2. If a faulty event producer issues conflicting events to different non-faulty EPA replicas, at most one of the events could be included for computing a decision.

Proof of P1: To accept a decision, an event consumer must collect 2f + 1 matching decision messages from different EPA replicas. There are at least f + 1 of the decision messages are sent by non-faulty EPA replicas, which constitutes the majority of the non-faulty EPA replicas.

Proof of P2: It is impossible for a non-faulty event consumer to accept a decision that is computed directly using two conflicting events. When a decision is accepted at an event consumer, it means 2f + 1 EPA replicas have produced exactly the same decision.

The only other scenario it must be considered is when two or more event consumers are targets of the decision from the same context partition CP(s). We prove that our framework satisfies P2 in this case by contradiction. Assume that a faulty event producer sends two conflicting events e1 and e2 to two different subsets of EPA replicas. Further assume that the decision accepted at one event consumer C1 included e1 in the computation of the decision, and the decision accepted at another event consumer C2 included e2 in the computation of the decision. According to our mechanisms, for C1 to accept the decision, it must have collected matched decision messages from a quorum R1 of 2f + 1 EPA replicas.

Similarly, for C2 to accept the decision, it must also have collected matched decision messages from a quorum R2 of 2f + 1 EPA replicas. There are only 3f + 1 EPA replicas in total, so R1 and R2 must intersect in at least f + 1 replicas. There are at least one of the replicas in the intersection of R1 and R2 must be non-faulty and a non-faulty EPA replica does not accept two conflicting events by definition, so it is impossible to have the decision accepted by conflicting events.

6.3 Implementation

Our BFT framework is also implemented with the interface provided by Esper, an open-source event stream processing library, to tolerate Byzantine fault. The BA cluster is implemented separate and easy for configuration. The event stream processing application is supported by Esper, which the event producers send stock events containing the most recent price to the EPA. The event consumers registered a listener in the EPA to listen to low price stock coming within the past 5 sec. For the same event context identifier and timestamp, a faulty event producer sends an event content (consists of the stock price and symbol) to some replicas and different event content to other replicas. These will cause the voting to fail at the event consumer and trigger a sync process for all replicas. The sync process filters out these conflict events based on the algorithm described in section 6.2.3 and the rest of the good events will be re-executed. We modified the application to use external event timestamp so the events inserted in a context partition are determined.

6.4 Performance Evaluation

The evaluation of the runtime performance of our mechanisms is carried out in a LAN testbed consisting of 14 HP BL460c blade servers and 18 HP DL320G6 rack-mounted servers connected by a Cisco 3020 Gigabit switch. Each BL460c blade server is equipped with two Xeon E5405 processors and 5 GB RAM, and each DL320G6 server is equipped with a single Xeon E5620 processor and 8GB RAM. All the servers run the 64-bit Ubuntu Linux server OS. The event producers and event consumers are deployed in different servers in the LAN testbed.

The performance of our BFT framework is measured for:

Voting latency. It is the delay caused by BFT voting at the event consumer, starting when a consumer receives the first alert, and ending when it receives the last one that make it 2f + 1 matching decisions.

Synchronization latency. It is the latency for completing a round of state synchronization, starting when a replica receives the first DECISION_FAIL message from the event consumer, and ending when it completes the sync and sends new decision to the consumer.

During the experiments, we varied the replication degree for EPA as well as the BA cluster from 1 to 4. The results are shown in Figure 22. As can be seen in Figure 22, the median value of the latency data collected for voting as well as the median value of the latency date collected for state synchronization increases linearly with the replication degree, which is expected because the number of sync messages collected for voting and state synchronization increases linearly with the replication degree. We use median for

measuring the latency because of unpredictable long tail in the distribution of the latency measurement, as shown in Figure 23.

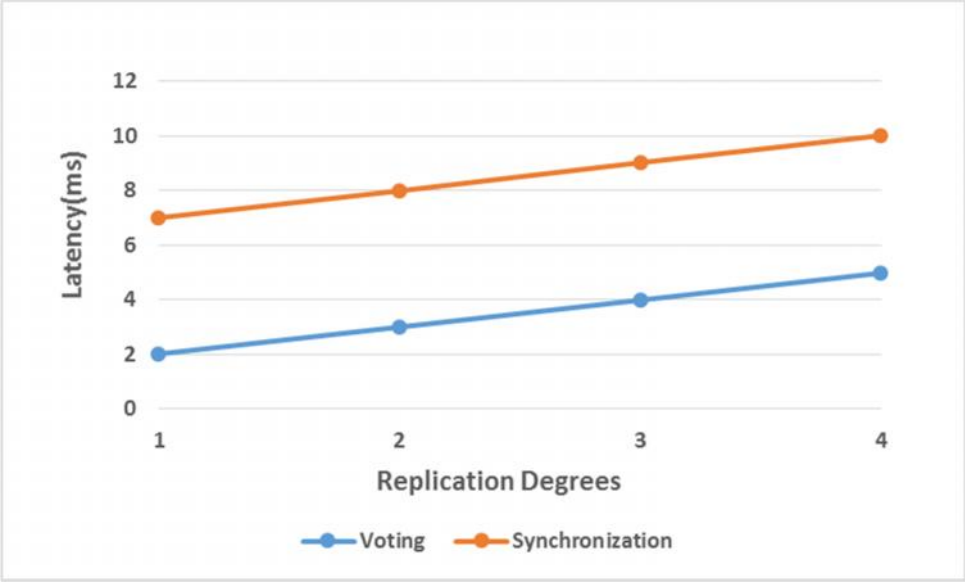


Figure 22 Latency versus replication degree for voting and synchronization.

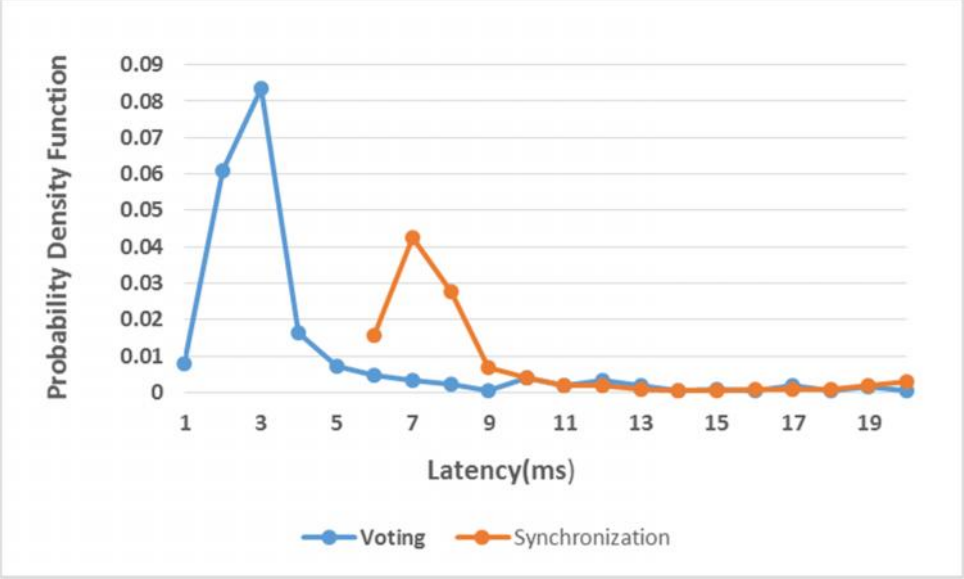


Figure 23 Probability density function of the latency for voting and synchronization.

The measurement results shown in Figure 23 confirms that with high probability that voting and state synchronization can be completed within a few milliseconds, occasionally we do see relatively large delays. However, the occasional large delays may be attributed to the limited threading, the asynchrony in communication and processing in our testbed instead of our mechanisms. The delays could also be attributed to runtime overhead for threading notification supported in the language for implementing our mechanisms. For systems that require tight realtime deadlines, special networking equipment and realtime operating systems are expected to avoid such unpredictable delays.

CHAPTER VII

BFT FOR SERVICES WITH CRDTS

In this chapter, we carefully analyzed the potential threats towards the service systems constructed with CRDTs. Based on the threat analysis, we presented a lightweight BFT framework together with associative mechanisms which could prevent the analyzed threats compromising the service integrity.

The system module is client-server architecture, where the data structure in the server is constructed purely by one or more CRDTs. The server is replicated and one or more clients broadcast their requests to the server replicas and wait for replies back.

7.1 Thread Analysis

In this section, we analyzed the potential threats that could compromise the integrity of the replicated service system as below.

7.1.1 Threats from a Faulty Server Replica

A faulty replica may not respond to a client request. E.g., the faulty replica may fail to send a reply due to a crash fault or refuse to respond to the client due to a malicious fault.

A faulty replica may lie about its execution status by sending replies with inconsistent results from the replica. E.g., the faulty replica may not even execute a request but send a reply to the client, it sends a reply but does not perform the operation intended by the client, or it sends a reply after correctly executing the client request but the result of the reply is inconsistent with that of the actual execution. Such threats can be mitigated by using sufficient server replicas and voting on the received replies from different replicas at the client. We don't handle the threats incurred by a non-malicious fault like a server error handling issues that some servers may be designed not to report exceptions to clients which may assume the execution of their requests is successfully completed.

7.1.2 Threats from a Faulty Client

A faulty client may attempt to attack other clients indirectly by causing divergence of the non-fault replicas' states through sending conflicting requests to different replicas. The state divergence problem caused by such threats can be mitigated by periodic and on-demand state synchronization (elaborated in section 7.2.3) among the replicas.

A faulty replica could send malformed requests. Like other BFT research works, we don't handle such threats in our research and these threats can be addressed by conventional security mechanisms [63] [64].

7.2 System Model and Solution Design

7.2.1 System Model Analysis

Our system module is client-server architecture, where the data structure in the server is constructed purely by one or more CRDTs. E.g., a shopping cart application with its cart data constructed by a U-Set data type.

A client may intent to perform two different types of operations at the server:

Update operations. Operations to request updating the state stored at the server.

Read-only operations. Operations to request retrieving the state of the server.

All update operations towards the state of the server are commutative and the eventual state after performing a set of operations is irrelevant to the operations' order.

Our system operates under Byzantine fault model that when the server is replicated, a faulty client or a fault server replica could send conflicting requests to different non-faulty server replicas. Basically, if two requests issued by a faulty entity have the same timestamp and the same client identifier but different operations, the two requests are considered conflicting. If non-faulty replicas update their states based on processing conflicting requests, unlike systems operate under crash-fault model that the states will be eventually consistent among all non-faulty replicas when clients stop sending new requests for sufficiently long period of time, the replicas under Byzantine fault model might never get their states converged. Thus, our system module includes a BA cluster to perform state synchronization to bring all non-faulty replicas' states converged.

7.2.2 Assumptions and Requirement

We assume the application server is replicated by 3f + 1 server replicas to tolerate up to f faulty replicas. None of the replicas plays the primary role since selecting one replica as the primary is not necessary for the server cluster.

We assume the BA cluster is composed of 3f + 1 ordering nodes and all the nodes run the same BFT algorithm (which could an adapted version of any traditional BFT algorithm). One of the replicas is selected as the primary in charge of total ordering of incoming messages to the BA cluster. The BA cluster is used for performing a BA operation on the application client requests before they are delivered and executed at the application server replicas. A BA operation can ensure that all non-faulty application server replicas deliver the same set of requests in the same total order.

We assume all requests have a unique identifier and the identifier could be a sequence number which is monotonically increased between each pair of the communication entities. A response message contains the identifier that matches that of its corresponding request.

We assume that all messages exchanged from different entities are all protected by a security token which could be either a digital signature or message authentication code.

7.2.3 The Lightweight BFT Framework

In this subsection, we elaborated our BFT framework and associative mechanisms. Our lightweight BFT solution extends existing studies [4-7]. As for the mechanisms, we

do not use something similar that has been used in [65], which facilitates the discovery of missed requests and the detection of conflicting requests without explicit state synchronization due to the increased message size.

Our BFT framework includes two different modes: normal operation and state synchronization modes.

1) Normal Operation: in this operation mode shown in Figure 24, a client sends a request <REQUEST, o, t, c>δ_c to all server replicas, where o is an update/read-only operation, t is the timestamp, c is the identifier of the client, and δ_c is the security token (e.g., digital signature) for protecting the request. When a server replica receives a client request, it validates the message by verifying the security token and checking the timestamp for the received request. The timestamp can help a server replica to detect duplicate messages. If the request is invalid, the replica executes the operation in the request and sends the corresponding reply <REPLY, t, c, i, R>δ_i to the client. In such a replay, c is the client identifier, t is the timestamp of the corresponding request, i is the replica identifier, and R is the results of executing the requested operation. For an update request, the replica logs the request and a proof of execution record <t, c, d>, where d is the digest of the update request sent by the client c.

Once a client sends a request, it sets up a timer and waits for 2f + 1 matching replies from different server replicas with the same timestamp t, client identifier c, and results R. If the client can collect sufficient matching replies before timeout, it accepts and delivers the reply and then resumes sending the next request; otherwise, it retransmits the request to all replicas.

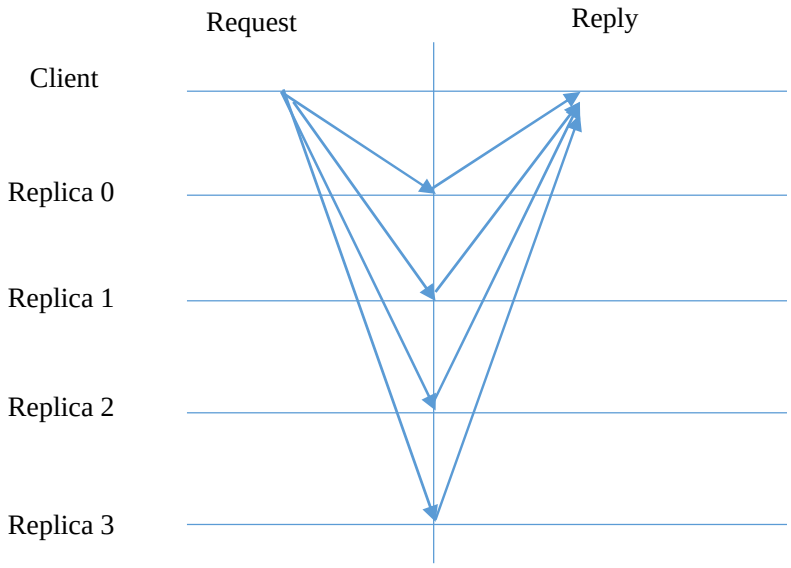


Figure 24 Normal Operation of the BFT mechanism with f = 1

2) State Synchronization:

Due to the potential threats that a faulty client or any fault replicas could send conflicting requests to different non-faulty server replicas, explicit rounds of synchronization on the replicas' states are necessary to achieve state convergence. In our BFT framework, we have two types of state synchronization:

Periodical synchronization. State synchronization is trigger every time a non-faulty replica has executed a predefined a number of requests since the last round of synchronization. This synchronization also helps on performing garbage collection for received requests and truncating logs maintained for executed update operations' history.

On-demand synchronization. State synchronization may also be triggered by a client's sending a synchronization request. This could happen when a faulty client issues conflicting requests to different server replicas, which could cause a non-faulty client may

not be able to collect sufficient (2f + 1) matching replies to its request even if after a set number of retransmissions. If a client demands a round of state synchronization, it has to provide the collected mismatched replies (2f + 1 or more) to its request as the proof that the synchronization is necessary.

Upon entering a round of state synchronization, a replica sends an Agreement request <AGREEMENT, b, i, O>δ_i to the BA cluster and suspended executing requests received from clients. In the Agreement request, b is the unique identifier of a specific synchronization round, i is the replica identifier, and O is a list of executed update operations' history entries, where each entry is a proof-of-execution record <t, c, d> for an update operation that is logged at the replica i. The non-faulty server replicas may have different O contained in their Agreement requests since they are allowed to execute requests from the clients in parallel.

The BA cluster totally orders all received Agreement requests and send them to all non-faulty server replicas so that all the replicas can receive the same set of Agreement requests in the same order. Upon receiving the first 2f + 1 totally ordered Agreement requests, a replica computes two lists of operations:

Super Set Ops. It is a superset of the update operation histories contained in the 2f + 1 totally ordered Agreement requests.

Undo List. A list of update operations that a replica has already executed but need to rollback the state changes they made. For a non-faulty server replica, an operation in this list is never able to be added in the *Super Set Ops.*

How to compute *Super Set Ops* and *Undo List* is described as below:

If an operation identified by timestamp t and client identifier c is included in operation histories, and all the records with identical t and c have the same digest d (no conflicting records are detected), the operation record is included in the Super Set Ops. If two or more records for an update operations are detected with identical t and c but different digest d, they are due to conflicting requests sent by a Byzantine faulty client. However, if at least f + 1 matching records for the operation can be found in the operation histories in the received 2f + 1 totally ordered Agreement requests, the operation is included in *Super Set Ops*. The faulty client which issued the corresponding conflicting requests is blacklisted and rejected to accept any future request. The mechanism can guarantee that at most one operation of the conflicting records can be included in *Super Set Ops.* If an executed update operation with two or more records cannot collect any f + 1 matching records in the received operation histories, the operation is included in *Undo List*.

After a replica has processed all the 2f + 1 Agreement requests, it examines each operation in *Super Set Ops* since it is possible that one or more operations included in this list have not been executed. In this case, the replica looks for its local storage to check if it has received the update requests containing those unexecuted operations. For the unexecuted operations which can be found in the local storage, the replica executes them immediately. For the unexecuted operations which cannot be found in the local storage, the replica derives a *To Do List* for those operations and asks for retransmission of each operation in the *To Do List* from other replicas which sent the Agreement requests containing the record of the operation. Upon receiving a retransmitted request, a replica retransmits the client request containing the required operation if it can find a received client request containing an operation matching the record <t, c, d> in the retransmitted

request. Once the replica asking for retransmission receives a valid client request with the expected operation, it executes the operation immediately and deletes the corresponding entry in *To Do List*.

To keep state consistency, each replica needs to performs an undo operation for each operation included in *Undo List* to rollback the state changes caused by the executed operation. With such a mechanism, a bad update operation would have no impact on other good updates regardless of their relative execution order. It greatly reduces the negative impact of executing conflicting requests at different non-faulty replicas.

When a replica has executed all operations in *Super Set Ops* and has rolled back all operations in *Undo List* that it has executed, its operation history would consists of only those in *Super Set Ops*, as shown in Figure 25. At the end of this computation, the replica takes a checkpoint of its state (made by the operations only in *Super Set Ops*) and notify the checkpoint to all other replicas. Once the same checkpoint is confirmed installed in 2f + 1 distinct server replicas by receiving checkpoint notifications and comparing the checkpoints' digest contained in the messages, the replica can garbage collect all executed client requests and truncate the history log to the checkpoint. Then the replica can resume execution of client requests.

Figure 25 Build *Super Set Ops* based on each replica's operation history.

7.2.4 Proof of Correctness

Our lightweight BFT algorithm can guarantee the replicated service satisfies the following properties:

P1. If a client sends a request, it will eventually receive 2f + 1 matching replies and eventually get this request executed at all non-faulty replicas. (This property ensures eventual state consistency of non-faulty server replicas.)

P2. The reply to a read-only request reflects the state changes made by all update requests that causally precede it. (This property ensures causality of the operations of the system, i.e., if an update request causally precedes a read-only request, the reply to the read-only request must reflect the state changes made by the update request.)

P3. If a client sends conflicting update requests to different non-faulty server replicas, the state changes made by at most one of the requests will be propagated to other non-faulty clients. (This property with P2 prevents cascading rollbacks when a faulty client that has sent conflicting requests is detected.)

Proof of P1: A server replica executes a client's request immediately unless it is in the middle of a round of state synchronization. A non-faulty client would repeatedly retransmit its request to all replicas until it has collected 2f + 1 matching replies. The synchronization round will eventually terminate when the replica has collected 2f + 1 totally ordered agreement messages from different replicas.

For an update request, as long as all non-faulty server replicas received and executed the request, the client is guaranteed to receive matching replies from a quorum R1 of 2f + 1 replicas. During the next round of state synchronization, every non-faulty

replica would collect agreement messages from a quorum R2 of 2f + 1 replicas. R1 and R2 must intersect in at least f + 1 replicas and one of them must be non-faulty. This non-faulty replica ensures the dissemination of the update request to all non-faulty replicas. For a read-only request, a client might not be able to collect 2f +1 matching replies initially if a faulty client has issued conflicting requests previously. However, the non-faulty client that issued the read-only request will always be able to collect 2f +1 replies from different replicas. If there exist nonmatching replies from the set of replies it has collected, the client would demand a round of state synchronization. When the round of on-demand state synchronization is completed, the client is guaranteed to receive 2f + 1 matching replies because the states of non-faulty replicas are converged.

Proof of P2: An update request causally precedes a read-only request issued by a client c in either of the two scenarios:

1) The update request is issued by the same client c prior to the read-only request.

2) The update request is issued by another client c' and its state changes are read by another read-only request ro1 issued by the same client c prior to the current read-only request ro2.

In the first scenario, the client c must have received matching replies from a quorum R1 of 2f + 1 replicas for the update request prior to the sending of the read-only request. Subsequently, the client must also have received 2f + 1 matching replies from a quorum of replicas R2 for the read-only request. Because there are 3f + 1 server replicas, R1 and R2 must intersect in at least f + 1 replicas, which means that at least one non-faulty replica has executed first the update request and then the read-only request. This ensures the proper causality of requests processing.

For the second scenario, because we assume that the reply to ro1 reflects the state changes caused by the update request issued by client c', a quorum R3 of 2f + 1 server replicas must have executed both the update request and the read-only request ro1. For the next read-only request ro2, client c would wait until it has collected a set of 2f + 1 matching replies. This means that there exists a quorum R4 of 2f + 1 replicas that executed ro2. R3 and R4 must intersect in at least f + 1 replicas, and one of them must be a non-faulty replica. This non-faulty replica ensures that the causal dependency is respected. This proves that our BFT mechanisms ensure property P2.

Proof of P3: The state changes at the server replicas may be propagated to a non-faulty client when the client issues a read-only request. If prior to this read-only request, a faulty client issued conflicting update requests to different replicas, at least 2f +1 of the replicas have reached consistent state. Otherwise, they could not have generated matching replies to the later read-only request since the client would not accept a reply to its read-only request until it has collected 2f + 1 matching replies from different replicas.

We prove the property P3 by contradiction. Assume that the state changes caused by two conflicting update requests with identical timestamp t and client id c, rw1 and rw2 respectively, have been propagated to one or more non-faulty clients. This implies that a quorum R1 of 2f +1 server replicas have executed rw1, and a quorum R2 of 2f +1 server replicas have also executed rw2. The two quorums R1 and R2 must intersect in at least f +1 replicas, which means one of them must be a non-faulty replica. It is impossible that a non-faulty replica has accepted and executed both rw1 and rw2 that contain the same timestamp t from the same client c.

7.3 Implementation

We have implemented our BFT framework and a test application (a shopping cart application constructed by using the U-Set [7]). A U-Set type is one of the common and simple CRDTs and it is comprised by two grow-only sets: the adding set and the tombstone set. The adding set is used to store objects for set “add” operations while the tombstone set is used to store objects for set “remove” operations. The state constructed by the U-Set type is the objects in the adding set excluding the objects in the tombstone set. It is apparent that the “add” and “remove” operations are commutative. In addition to the “add” and “remove” operations, the shopping cart application also allows a client to query items in its shopping cart.

We use the UpRight library [22] as the BA cluster described in our BFT framework. UpRight is a Java-written open source library implementing Zyzzyva [11], a well-known BFT algorithm. This library enables separating the servers comprising the BA cluster from servers of the application, which helps our implementation with less development cost. The library also offers highly efficient BFT total ordering service during normal operation mode.

We configure the application server is replicated with 3f + 1 replicas to tolerate f faulty ones and we configure f = 1. The BA cluster is also configured with 3f + 1 ordering nodes to tolerate f faulty ones and we also configure f = 1. Each node in the BA cluster runs the configured UpRight library and one of the nodes is configured as the primary leading the total ordering process. Once a request sent to the BA cluster is totally ordered,

the totally ordered request with ordering information is sent to all application server replicas.

7.4 Performance Evaluation

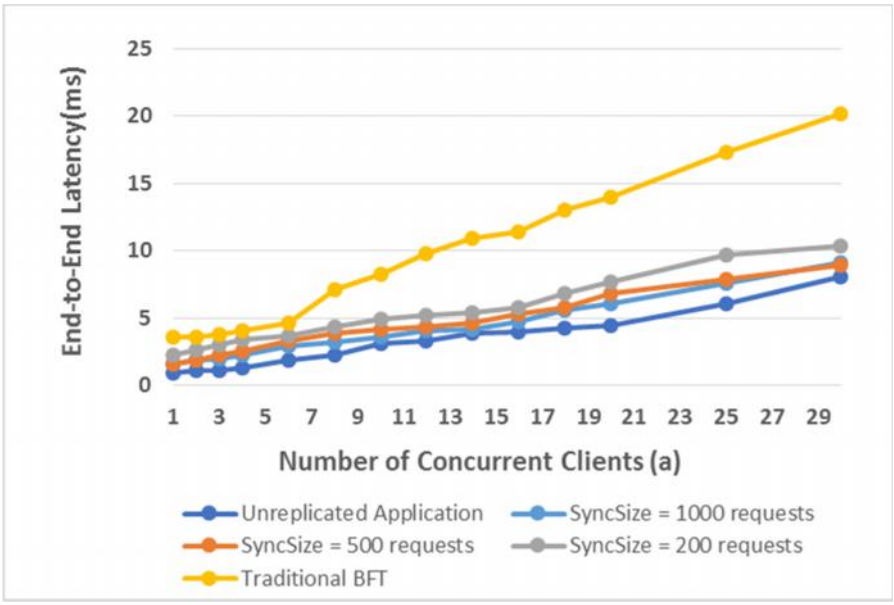
The performance evaluation is carried out in a LAN testbed that consists of 14 HP BL460c blade servers and 18 HP DL320G6 rack-mounted servers connected by a Cisco 3020 Gigabit switch. Each BL460c blade server is equipped with two Xeon E5405 processors and 5 GB RAM, and each DL320G6 server is equipped with a single Xeon E5620 and 8GB RAM. All the server nodes in the LAN testbed run a 64-bit Ubuntu Linux server version OS.

We experimented with three different system configurations:

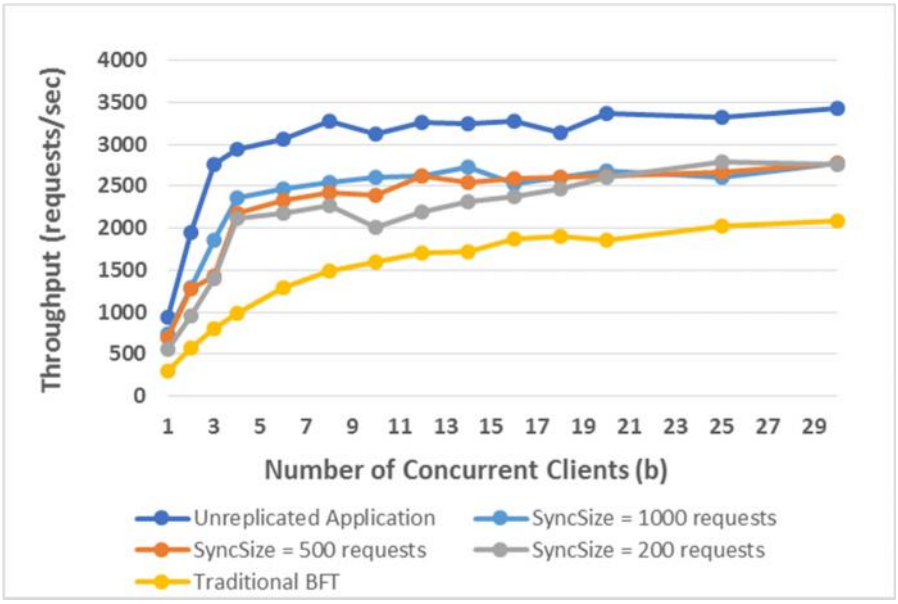
Unreplicated Application. The server in shopping cart application is not replicated and the application is not protected by our lightweight BFT.

Lightweight BFT. The application server is replicated and the application is protected by our lightweight BFT framework. The experiments are carried out when the system is configured by the predefined number of executed requests for entering each state synchronization round 200, 500 and 1000.

Traditional BFT. The application server is replicated and all requests are totally ordered before they are executed. To fairly compare the performance of system configured for *Lightweight BFT* and *Traditional BFT*, we allow all the requests are concurrently executed at each server replica side.



(a)



(b)

Figure 26 End-to-end latency and throughput with different number of concurrent clients.

For all configurations, we measure the end-to-end latency at the client side and measure the throughput at one of the server replicas. The results are shown in Figure 26

((a) – (b)). To simulate some substantial computation for each request at the server to have the experimental data distribution trend clearer, we have each operation have large enough execution time around by injecting 0.5ms computation loops. Compared with the baseline (the unreplicated configuration) shown in Figure 26(a), the average end-to-end latency of our BFT framework is a little bit higher but much less than that of the traditional BFT configuration. This is probably because the additional latency incurred by each request in our BFT framework is almost constant (about 1-2ms) regardless of the number of concurrent clients while that in the traditional BFT configuration increases roughly linearly with the number of concurrent clients.

As for the throughput results shown in Figure 26(b), compared with the baseline, our BFT framework reduced the throughput by about 20% but much better than the performance of the traditional BFT configuration which reduces the throughput by about 40%. It can also be observed that the throughput for different synchronization size (200, 500 and 1000 requests per synchronization) remain roughly the same. This is probably because the latency incurred by state synchronization is not apparently larger than that incurred by normal execution of requests. We measured the cost of state synchronization and the results are shown in Figure 27. The larger the number of the records (synchronization size) contained in each exchanged operation history, the higher the cost of the state synchronization. During the state synchronization, the throughput is reduced in average due to the process of the request execution is suspended. For a configured synchronization size of 200 requests, the system has less synchronization overhead but higher synchronization frequency than that of the synchronization size of 1000 requests.

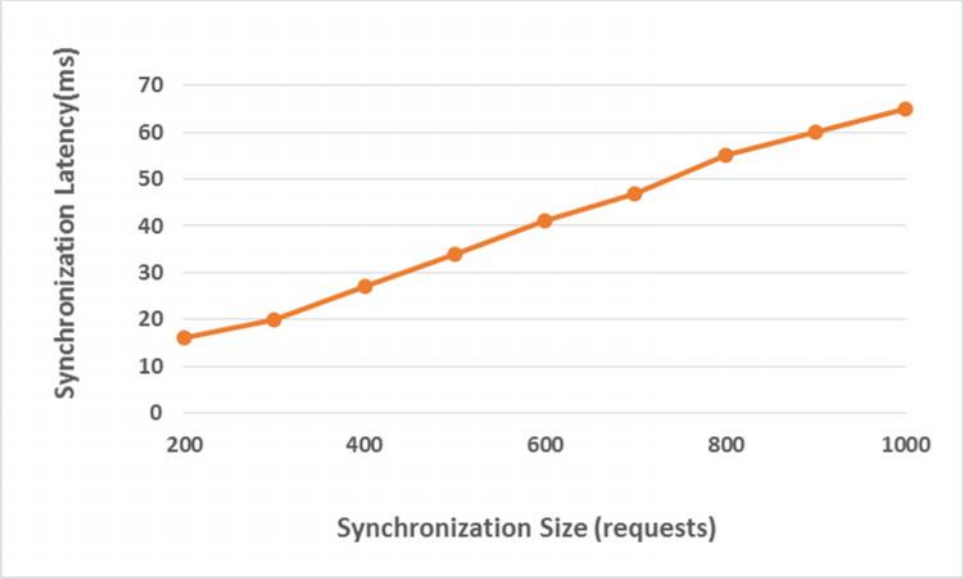


Figure 27 State synchronization latency.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

In this chapter, the major research contributions in this dissertation are summarized and some future work is discussed.

The major contributions of this research work include:

- Providing guidance for design of BFT framework through exploiting application semantics,
- Apply BFT model to applications with different semantics, e.g., WSBA, WSAT, session oriented middle tier and event stream processing,
- Using Byzantine agreement as a service and apply it to most of our research,
- A set of BFT mechanisms to enable concurrent execution of independent requests on systems constructed by CRDTs,

The future work will focus on achieving system throughput by using non-blocking state synchronization on system with CRDTs and implementing Undo List for all different major CRDTs.

8.1 Conclusion

There are a lot works on BFT that have been developed in the past several decades [1] [8] [11] [12] [13] [22]. The most common approach is state-machine replication with a Byzantine agreement algorithm to ensure the total ordering of all incoming requests from various clients to a replicated service. Such an approach would require that all requests are executed sequentially according to the total order established. Unfortunately this would render the technology impractical for many application services because it would impose severe runtime overhead.

In this dissertation research, we aim to overcome this limitation by designing BFT solutions based on exploiting application semantics. This approach could enable concurrent execution of independent requests. We designed a set of BFT mechanisms for each application in WSBA, WSAT, session oriented multi-tier and event stream processing. Performance evaluation shows the approach is promising for these applications. However, before designing a BFT solution for a specific application, we need intimate knowledge on the application design and implementation, which implies for every different application, we need to design a different BFT solution. It could require too much development effort on the application analysis and solution design.

Recent research work suggests that a service may be constructed by using Conflict-free Replicated Data Types (also referred to as commutative or convergent replicated data types, or CRDTs in short) for highly efficient optimistic replication with the crash-fault model [4] [5] [6] [7]. Due to using CRDTs to construct service state, all operations through the service are commutative and requests can be executed concurrently without the need to

know the semantic of the target applications. The proposed solution [7] only handles crash faults and cannot convert a crash fault model to a Byzantine fault model when the service exists a faulty identity. In this dissertation research, we applied Byzantine fault model to the services or applications constructed by CRDTs and designed a set of BFT mechanisms to handle Byzantine faults.

8.2 Future Work

We have shown that BFT with using CRDTs indeed can significantly improve the performance of the replicated system. However, we expect to do more in future work:

(1) Non-blocking state synchronization. For the state synchronization mechanism described in Section 7.2.3, a replica would stop executing new requests until the synchronization for the round is completed. This blocking behavior is obviously undesirable. We can have the performance even better if we can modify the mechanism to avoid blocking as described below.

The state synchronization mechanism can be modified to avoid blocking. For non-blocking state synchronization, a replica would continue accepting and executing new valid requests during the synchronization. After it has collected 2f + 1 totally ordered agreement messages, the replica would proceed to building the superset of operations as usual. Because the replica does not block after sending its agreement message, it may have executed one or more update requests that are not included in the superset, Super Set Ops.

The replica needs to identify such operations and temporarily undos them before it takes a checkpoint. Once it takes a checkpoint, the replica re-executes the operations again.

(2) Undo executed operations towards CRDTs. In future work, we plan to implement Undo List for the major CRDTs mentioned in the paper [7]. They could be any of the following:

- Replicated counters: G-Counter, PN-Counter, Non-negative,
- Registers, or mutable shared variables: LWW-Register, MV-Register,
- Sets: G-Set, 2P-Set, U-set, PN-Set, OR-Set,
- Graphs: 2P2P-Graph, Add-only monotonic DAG.

For an application constructed by a specific CRDT, to perform an undo on an already executed operation, we want to revert the change made by an operation anytime without sacrificing the performance. It should be realistic to achieve this if the undo operation is evaluated or designed commutative with other operations towards the CRDT application.

BIBLIOGRAPHY

[1] R. Kotlan and M. Dahlin, "High throughput byzantine fault tolerance," in *Proceedings of International Conference on Dependable Systems and Networks*, 2004.

[2] T. Freund and M. Little, "Web Services Business Activity, Version 1.1," OASIS Standard, April 2007.

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

[4] C. Baquero and F. Moura, "Using structural characteristics for autonomous operation," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 4, pp. 90-96, 1999.

[5] N. Preguiça, J. Marquès, M. Shapiro and M. Leţia, "A commutative replicated data type for cooperative editing," in *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, Jun 2009.

[6] H.-G. Roh, M. Jeon, J.-S. Kim and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354-368, 2011.

[7] M. Shapiro, N. Pregui, C. Baquero and M. Zawirski, "Conflict free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems, ser.*, Springer Berlin Heidelberg, 2011.

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

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

[10] M. Pease, R. Shosta and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM,* vol. 27, no. 2, pp. 228-234, 1980.

[11] R. Kotla, L. Alvisi, M. Dahlin, A. Clement and E. Wong., "Zyzzyva: Speculative byzantine fault tolerance," in *Proceedings of 21st ACM Symposium on Operating Systems Principles*, pp. 45-58, New York, NY, USA, 2007.

[12] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. Reiter and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, pp. 59-74, Brighton, UK, Oct. 2005.

[13] J. Cowling, D. Myers, B. Liskov, R. Rodrigues and L. Shrira, "Hq replication:A hybrid quorum protocol for Byzantine fault tolerance," in *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations*, pp. 177-190, Seattle, WA, November 2006.

[14] M. Feingold and R. Jeyaraman, Web Services Coordination, Version 1.1, OASIS Standard, July 2007.

[15] H. Erven, H. Hicker, C. Huemer and M. Zapletal, "The Web Services-BusinessActivity-Initiator (WS-BA-I) protocol: An extension to the Web Services-BusinessActivity specification," in *Proceedings of the IEEE International Conference on Web Services*, Salt Lake City, UT, July 2007.

[16] The Open Group, "Distributed Transaction Processing: The XA Specification,," February 1992.

[17] A. Almeida, J.-P. Briot, S. Aknine, Z. Guessoum and O. Marin, "Towards autonomic fault-tolerant multi-agent systems," in *Proceedings of the 2nd Latin American Autonomic Computing Symposium*, Petropolis, RJ, Bresil, 2007.

[18] A. Padovitz, A. Zaslavsky and S. W. Loke, "Awareness and agility for autonomic distributed systems: platform-independent and publish subscribe event-based communication for mobile agents," in *Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, 2003.

[19] "Esper," [Online]. Available: http://esper.codehause.org..

[20] O. Etzion and P. Niblett, "Event Processing in Action," Manning Publications, 2010.

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

[22] 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.

[23] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith, "The SecureRing group communication system," *ACM Transactions on Information and System Security*, vol. 4, no. 4, pp. 371-406, November 2001.

[24] D. Malkhi and M. Reiter, "Secure and scalable replication in Phalanx," in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, Lafayette, IN, 1998.

[25] L. E. Moser and P. M. Melliar-Smith, "Byzantine-resistant total ordering algorithms," *Journal of Information and Computation*, vol. 150, no. 1, pp. 75-111, 1999.

[26] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith, "The SecureRing group communication system," *ACM Transactions on Information and System Security*, vol. 4, no. 4, pp. 371-406, 2001.

[27] 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.

[28] C. Mohan, R. Strong and S. Finkelstein, "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*, pp. 89-103, Montreal, Quebec, Canada, 1983.

[29] W. Zhao, "A Byzantine fault tolerant distributed commit protocol," in *Proceedings of the IEEE International Symposium on Dependable, Autonomous and Secure Computing*, Columbia, MD, September 2007.

[30] H. Garcia-Molina, F. Pittelli and S. Davidson, "Applications of Byzantine agreement in database systems," *ACM Transactions on Database Systems,* vol. 11, no. 1, pp. 27-47, 1986.

[31] H. Zhang, H. Chai, W. Zhao, P. M. Melliar-Smith and L. Moser, "Trustworthy coordination of Web services atomic transactions," *IEEE Transactions on Parallel and Distributed Systems,* vol. 23, no. 8, p. 1551–1565, 2012.

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

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

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

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

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

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

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

[39] M. Steiner, G. Tsudik and M. Waidner, "CLIQUES: A new approach to group key agreement," in *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, pp. 380-387, Amsterdam, The Netherlands, May 1998.

[40] W. Zhao, "BFT-WS: A Byzantine fault tolerant framework for Web Services," in *Proceedings of the Middleware for Web Services Workshop*, pp. 89-96, Annapolis, MD, October 2007.

[41] H. Chai, H. Zhang, W. Zhao, P. M. Melliar-Smith and L. Moser, "Towardtrust worthy coordination of Web services business activities," *IEEE Transactions on Parallel and Distributed Systems,* vol. 23, no. 8, p. 1551–1565, 2012.

[42] A. Brito, C. Fetzer and P. Felber, "Multithreading-enabled active replication for event stream processing operators," in *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*, 2009.

[43] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, 1995.

[44] H. Zhang and W. Zhao, "Concurrent byzantine fault tolerance for software-transactional-memory based applications," *International Journal of Future Computer and Communication,* vol. 1, no. 1, p. 2012, 47–50.

[45] H. Chai and W. Zhao, "Byzantine Fault Tolerant Event Stream Processing," in *Proceedings of the 12th IEEE International Conferencce on Dependable Autonomic and Secure Computing (DASC)*, Dalian, China, 2014.

[46] Y. Tohma, "Incorporating fault tolerance into an autonomic-computing environment," *IEEE Distributed Systems Online,* vol. 5, no. 2, 2004.

[47] H. Chai and W. Zhao, "Byzantine fault tolerance for session-oriented," *Int. J. of Web Science,* vol. 2, no. 1/2, pp. 113-125, 2013.

[48] T. Distler and R. Kapitza, "Increasing performance in Byzantine fault tolerant systems with on-demand replica consistency.," in *Proceedings of the 6th*, 2011.

[49] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.,* vol. 37, no. 1, p. 42–81, Mar. 2005.

[50] E. A. Brewer, "Pushing the cap: Strategies for consistency and availability," *IEEE Computer,* vol. 45, no. 2, p. 23–29, 2012.

[51] R. Ramakrishnan, "Cap and cloud data management," *IEEE Computer,* vol. 45, no. 2, p. 43–49, 2012.

[52] C. Sun, X. Jia, Y. Zhang, Y. Yang and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.,* vol. 5, no. 1, p. 63–108, Mar. 1998.

[53] G. Oster, P. Urso, P. Molli and A. Imine, "Proving correctness of transformation functions in collaborative editing systems," INRIA, Rapport de recherche RR-5795, 2005.

[54] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Symp. on Op. Sys. Principles (SOSP)*, Oct 2007.

[55] D. Davis, A. Karmarkar, G. Pilz, S. Winkler and U. Yalcinalp, "Web Services Reliable Messaging, Version 1.1, OASIS Standard," January 2008.

[56] "Apache Kandula," [Online]. Available: http://ws.apache.org/kandula/.

[57] "Apache Axis," [Online]. Available: http://ws.apache.org/axis/.

[58] "Apache WSS4J," [Online]. Available: http://ws.apache.org/wss4j/.

[59] A. Nadalin, C. Kaler, R. Monzillo and P. Hallam-Baker, "Web Services Security: SOAP Message Security 1.1, OASIS Standard," November 2006.

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

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

[62] "Apache Derby," [Online]. Available: http://db.apache.org/derby/.

[63] J. P. Anderson, "Computer security technology planning study," USAF, Tech. Rep., 1972.

[64] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering,* Vols. SE-1, no. 2, p. 220–232, June 1975.

[65] W. Zhao, P. M. Melliar-Smith and L. E. Moser, "Low latency fault tolerance system," *The Computer Journal,* vol. 56, no. 6, p. 716–740, 2013.