Cleveland State University
EngagedScholarship@CSU

MSL
ACADEMIC ENDEAVORS

ETD Archive

2007

Byzantine Fault Tolerant Coordination for Web Services Atomic Transactions

Honglei Zhang
Cleveland State University

BYZANTINE FAULT TOLERANT COORDINATION FOR WEB SERVICES ATOMIC TRANSACTIONS

HONGLEI ZHANG

Bachelor of Science in Electrical Engineering

Taiyuan University of Technology

July, 2003

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

Nov 2007

This thesis has been approved for the

Department of **ELECTRICAL AND COMPUTER ENGINEERING**

and the College of Graduate Studies by

____

Thesis Committee Chairperson, Dr. Wenbing Zhao

____

Department/Date

____

Dr. Yongjian Fu

____

Department/Date

____

Dr. Lili Dong

____

Department/Date

To my loved wife Shuqiong and my mother...

ACKNOWLEDGMENTS

I would like to thank the following people:

Express my gratitude to Dr. **Wenbing Zhao** for all his abundantly help and full-of-insight support and guidance as my supervisor, and for the virtues I learned from him.

Deepest gratitude are also due to Dr. **Lili Dong** and Dr. **Yongjian Fu** for agreeing to be on my committee and for all of their assistance.

Special thanks to Dr. **Ye Zhu**, Dr. **Pong P. Chu**, Dr. **William L. Schultz** and Dr. **Dan Simon** for their elaborately prepared lectures which provide very helpful knowledges to enhance my view.

Gratitude to **Weijia Wang** for doing the English grammer checking.

Thanks to **Gang Tian**, **Qing Zheng**, **Hua Chai**, **Bo Chen** for their kind help and friendship.

Thanks also to all my friends, for sharing the literature and invaluable assistance.

I would also thank my wife, my mom and all my relatives. They supported me all the time.

BYZANTINE FAULT TOLERANT COORDINATION FOR WEB SERVICES ATOMIC TRANSACTIONS

HONGLEI ZHANG

ABSTRACT

This thesis describes a Byzantine Fault Tolerant Coordination framework for Web Service Atomic Transaction (WS-AT). In the framework, all core services, including transaction activation, registration, and completion & distributed commit, are replicated and protected by Byzantine fault tolerance mechanisms. The traditional Two-Phase Commit (2PC) protocol is extended by a Byzantine fault tolerant version that can tolerate arbitrary faults on the coordinator and the initiator sides, and some types of malicious faults on the participant side. To achieve Byzantine fault tolerance in an efficient manner, and to limit the types of malicious behaviors of the coordinator, a novel decision certificate is introduced. The decision certificate includes a signed copy of the participants' vote records, and it is piggybacked with all decision notifications to the participants for each participant to verify the legitimacy of the decision.

The Byzantine Fault Tolerance (BFT) mechanisms, together with the extended two-phase commit protocol, have been incorporated into an open-source framework supporting the standard Web services atomic transactions specification. Performance characterizations of the framework show that the implementation is fairly efficient.

Such a Byzantine fault tolerant coordination framework can be useful for many transactional Web services that require a high degree of security and dependability.

TABLE OF CONTENTS

Page

ABSTRACT v

LIST OF FIGURES ix

ACRONYM xi

CHAPTER

I. INTRODUCTION 1

II. BACKGROUND 4

2.1 Web Services Technology 4

2.1.1 Web Services 4

2.1.2 Web Service Technology standards 5

2.2 Web-Service Atomic Transaction 7

2.2.1 Web-Service Atomic Transaction Specification 7

2.2.2 Web-Service Atomic Transaction Model 11

2.3 Byzantine Fault Tolerance 13

2.3.1 Byzantine Fault 13

2.3.2 Byzantine Fault Tolerance 14

III. BYZANTINE FAULT TOLERANT COORDINATION FOR WEB SERVICES ATOMIC TRANSACTIONS 16

3.1 System Model 16

3.2 Threat Analysis 20

3.3 Byzantine Fault Tolerant Transaction Activation 22

3.4 Byzantine Fault Tolerant Registration 26

3.5 Byzantine Fault Tolerant Transaction Completion and Propagation . . . 27
IV. IMPLEMENTATION AND PERFORMANCE EVALUATION 35
4.1 Byzantine Fault Tolerant Coordination Framework for WS-AT . 36
4.1.1 Multicaster . . . 36
4.1.2 Piggybacking . . . 38
4.1.3 Voter and Vote Collector . . . 40
4.1.4 Byzantine Agreement Agents . . . 41
4.1.5 Security Handler . . . 47
4.2 Performance Evaluation . . . 49
V. CONCLUSION AND FUTURE RESEARCH . . . 57
5.1 Conclusions . . . 57
5.2 Future Work . . . 58
BIBLIOGRAPHY . . . 59
APPENDIX . . . 63
A. INSTRUCTION FOR KEY PAIR GENERATION . . . 64
B. USER GUIDE . . . 71

LIST OF FIGURES

Figure Page

1 The Web Service Architecture 5

2 WS-AT Completion Protocol 9

3 WS-AT 2PC Protocol 10

4 Travel Agent Example Coordinated by WS-AT 12

5 Framework of Initiator and Participants 17

6 Framework of the Coordinator 18

7 System Framework 19

8 BFT Activation Service 23

9 BFT Registration Service 27

10 BFT Completion and Coordination Services (1) 2PC 29

11 BFT Completion and Coordination Services (2) BFT Mechanisms . . 29

12 Multicaster Flowchart 37

13 Piggybacking Flowchart 39

14 Flowchart of BA-Pre-Prepare phase for Activation Service 43

15 Flowchart of BA-prepare phase for Activation Service 44

16 Flowchart of BA-commit phase for Activation Service 45

17 BA-pre-prepare Flowchart Coordinator and Completion Service . . . 46

18 Flowchart of BA-prepare or BA-Commit phases 46

19 Flowchart of the MySecHandler 48

20 Performance of Our Byzantine Agreement Algorithm 50

21 2PC Latency Comparison 51

22 End-to-End Latency from Initiator Side Comparison 51

23 End-to-End Latency from Client Side Comparison 52

24 2 Participants End-to-End Throughput Comparison 53

25 Distributed Commit Latency Comparison 53

26 2PC Latency Comparison 54

27 BA-Activation End-to-End Latency Comparison 54

28 Initiator Side End-to-End Latency Comparison 55

29 Client Side End-to-End Latency Comparison 56

30 End-to-End Throughput 56

ACRONYM

BFT Byzantine Fault Tolerance

2PC Two-Phase Commit

WS-AT Web Service Atomic Transaction

XML eXtensible Markup Language

HTTP HyperText Transfer Protocol

SOAP Simple Object Access Protocol

WSDL Web Services Description Language

UDDI Universal Description, Discovery and Integration

CHAPTER I

INTRODUCTION

In recent years, we have seen a wide-adoption of the Web services technology by major players on the Internet, such as IBM, Microsoft, Google, and Amazon. The Web services technology has become an extremely powerful tool to enable the large-scale enterprise applications and systems integrations, which makes it possible to achieve automated business-to-business interactions without human intervention. Most of the Web services are transactional, in which most of them require a high degree of security and dependability. Furthermore, a particular concern is the data consistency among the participants of atomic transactions.

The Two-Phase Commit (2PC) protocol [1] is a standard distributed commit protocol [2] used in the distributed transactions [4, 21, 22] to achieve atomicity of the actions taken by the participants. However, the 2PC protocol is designed with the assumptions that the coordinator and the participants are only subject to benign faults and that the coordinator can be recovered quickly. Consequently, this protocol does not work if the coordinator is subject to arbitrary faults, also referred to as Byzantine faults [3], because a faulty coordinator might send the conflicting decisions to different participants to prevent them from terminating the transaction atomically (*i.e.*, some participants may commit the transaction, while others abort it). There are also a number of system-level works on fault tolerant TP monitors, such as [12, 15]. However, they all use a benign fault model. Such systems do not work if the coordinator is subject to intrusion attacks. We have yet to see other system-level work on Byzantine fault tolerant TP monitors.

The problem of BFT distributed commit for atomic transactions has been of research interest in the past two decades [4, 18]. Mohan *et al.* in [4] first studied this problem and provided a solution by integrating the Byzantine agreement and the 2PC protocol. Most concretely, the second phase of the 2PC protocol is replaced by a Byzantine agreement phase. This can prevent the faulty coordinator from disseminating conflicting messages to different participants without being detected. However, this approach is not practical because of three main deficiencies. First, it involves all members of the root cluster in the Byzantine agreement phase for each transaction. Therefore, the overhead of reaching a Byzantine agreement could be high if the size of the cluster is large. Second, this method can only protect the members in the root cluster, but not the subordinate coordinators and the participants outside. Third, it requires the participants in the root cluster to have the knowledge about all other participants in the same cluster, which prevents dynamic propagation of the transaction. Our work, on the other hand, requires a Byzantine agreement only among the coordinator replicas and hence, allows dynamic propagation of transactions. Rothermel et al. [18] addressed the challenges of ensuring atomic distributed commit in open systems where participants may be compromised. However, [18] assumes that the root coordinator is trusted. This assumption negates the necessity to replicate the coordinator for Byzantine fault tolerance. Apparently, this assumption is not applicable to Web services applications.

The work closest to ours is Thema [16], which is a BFT framework [19, 20, 27] for generic multi-tiered Web services. Even though some of the mechanisms are identical, our work contains specific mechanisms to ensure atomic transaction commitment.

In this thesis, we carefully analyze the threats to atomic commitment of distributed transactions and evaluate strategies to mitigate such threats. The Byzantine agreement is only carried out among the coordinator replicas, which avoids the problems in [4]. The Byzantine agreement algorithm used in our framework is adapted from the Byzantine Fault Tolerance (BFT) algorithm described in [5] because of its efficiency. That BFT algorithm is originally designed to ensure completely ordered atomic multicast for requests to a replicated state server. A number of modifications to the algorithm have been made so that it fits the problem of atomic distributed commit. The most crucial change is made to the first phase of the BFT algorithm, where the primary coordinator replica is required to use a decision certificate, which is a collection of the registration records and the votes it has collected from the participants, to back its decision on the transaction's outcome. The use of such a certificate is essential to enable a correct backup coordinator replica to verify the primary's proposal. This also limits the methods that a faulty replica can use to hinder atomic distributed commit of a transaction.

CHAPTER II

BACKGROUND

This chapter will provide readers with the necessary background knowledge to better understand my thesis.

2.1 Web Services Technology

2.1.1 Web Services

In this section, we introduce the concept of the Web services and the basic building blocks of the Web services platform. There is no universal definition for the term Web services; in fact its interpretation varies drastically. Web services can be loosely defined as any type of services offered over the World Wide Web. However, only the services enabled by the Web services technology are referred to as Web services by many researchers and practitioners. In this chapter, we use the latter interpretation. The Web services technology refers to the set of standards that enable automated machine-to-machine interactions over the Web. The corner stones of the Web services technology include eXtensible Markup Language (XML)

[6], HyperText Transfer Protocol (HTTP), Simple Object Access Protocol (SOAP) [7], Web Services Description Language (WSDL) [8], and Universal Description, Discovery and Integration (UDDI) [9]. From an architectural point of view, the Web services platform consists of Web services providers, Web services consumers, and the UDDI registries that broker the providers and the consumers, as shown in Figure 1.

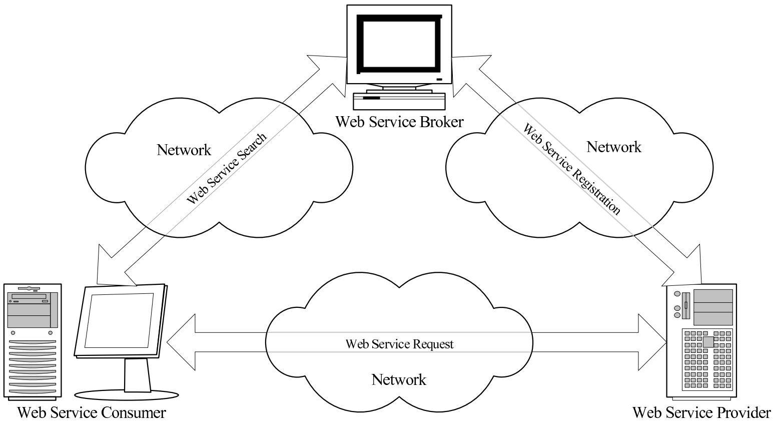

Figure 1: The Web Service Architecture

2.1.2 Web Service Technology standards

eXtensible Markup Language

XML is designed to facilitate self-contained, structured data representation and transfer over the Internet. It allows users to define their own tags, making it easily extensible. XML messages enable different applications to communicate with each other over the network using a variety of transport-level protocols such as HTTP and SMTP. To invoke a Web service, a user only needs to send an XML request message to the Web services provider. The provider will then send an XML reply message back containing the results requested by the user. Typically, the XML messages must conform to the SOAP standard.

Simple Object Access Protocol

SOAP, a communication protocol for exchanging messages over the Internet, provides a standard modular packaging model, a data encoding method and a way to perform remote procedure calls (RPCs) [26]. SOAP is easy to use and is easily extensible due to its use of XML as its message format. Like many public-domain application-level protocols, such as SMTP, a SOAP message contains a SOAP Envelop and a SOAP Body. A SOAP message often contains an optional SOAP Header element and a Fault element if an error is encountered by the sender of the SOAP message.

Web Service Description Language

WSDL provides a structured approach to describe a Web service based on an abstract model. For each Web service, the corresponding WSDL document specifies the available operations, the messages involved in these operations, and a set of endpoints to reach the Web service. WSDL is also extensible through the use of XML. In particular, it enables the binding of multiple different communication protocols and message formats.

Universal Discovery Description and Integration

The UDDI registry service acts like yellow pages for business providers and consumers. Business owners can publish their Web services to the UDDI registry; their partners and consumers can then locate the Web services in need and obtain detailed information by searching the registry. There are three main components in UDDI, commonly referred to as White Pages, Yellow Pages and Green Pages. The White Pages provide Web service provider's information, such as name, address, contact information and identifiers. The Yellow Pages describe industrial categories based on standard taxonomies. The Green Pages detailed present technical information regarding the Web services. The UDDI also support several methods of searching, e.g., search by service provider's location, by specified service types, etc...

2.2 Web-Service Atomic Transaction

2.2.1 Web-Service Atomic Transaction Specification

WS-Atomic Transaction builds on top of the Web-Service Coordination, which defines an extensible framework providing service protocols to coordinate activities in the distributed applications. At the end of each transaction, the coordinator will be asked to commit the proposed result. This commitment has "*all-or-nothing*" property. If they all vote "*yes*", which means the transaction has been executed successfully, then the coordinator commits the action and upon final agreement, the tentative transactions will be visible to all others. Otherwise, if even one participant votes to abort or does not response to the transaction, then the coordinator has to abort. This abort decision will make the tentative transaction appears as it had never happened.

Each WS-Atomic Transaction is modeled to have three actors, An Initiator (a special participant), A Coordinator and a few Participants.

The Initiator is responsible to start and terminate a transaction.

The Coordinator provides the following services:

Activation Service: At the beginning of a transaction, the initiator invokes the Activation Service to create a coordinator object, which will then generate a new coordination context for the transaction and return it to the initiator. The coordination context contains a unique transaction identifier and an endpoint reference [23] for Registration Service. This coordination context will be included in all request messages within the transaction boundary.

Registration Service: All participants including the initiator will register their endpoint references for other associated participant-side services. These endpoint references will be used by the coordinator to contact them during the two-phase commit protocol.

Coordinator Service: This service is responsible for propagating messages in the 2PC protocol to ensure atomic commit of a distributed transaction. This service will be invoked when it receives a request from the Completion Service and then it will start the 2PC with all participants excluding the initiator. The participants have obtained the endpoint reference of the Coordinator Service during the registration step.

Completion Service: This service is used to manage the distributed commits by the start notification from the transaction initiator. The Completion service, together with the CompletionInitiator service on the participant side, implements the WS-AT completion protocol. The endpoint reference of the Completion Service is returned to the initiator during the registration step, the same as Coordinator Service.

The following services are from Participants.

CompletionInitiator Service: This service is provided by the transaction initiator to start a distributed commit and to obtain the final outcome of the transaction, as a part of the Completion protocol.

Participant Service: This service is invoked by the coordinator to solicit votes from, and to send the transaction outcome to the participants according to the two-phase commit protocol.

The WS-AomicTransaction specification defines two protocols, Completion protocol and Two-Phase Commit (2PC) protocol.

Completion Protocol

The completion protocol occurs between the initiator and the coordinator to initiate a commit action by sending requests to the coordinator to either commit or abort the transaction and signal the final result. The commit request is sent to the coordinator when requested by the initiator. An instance of the Two-Phase Commit (2PC) protocol to carry out the agreement for the atomic transaction will then be launched by the coordinator. After the 2PC, a notification of the final outcome of the transaction will be sent to the initiator by the coordinator accordingly (*i.e.*, Committed or Aborted). However, if the request from the initiator is Rollback instead, the coordinator will then abort the transaction directly. Figure 2 below shows the completion protocol in an abstract manner.

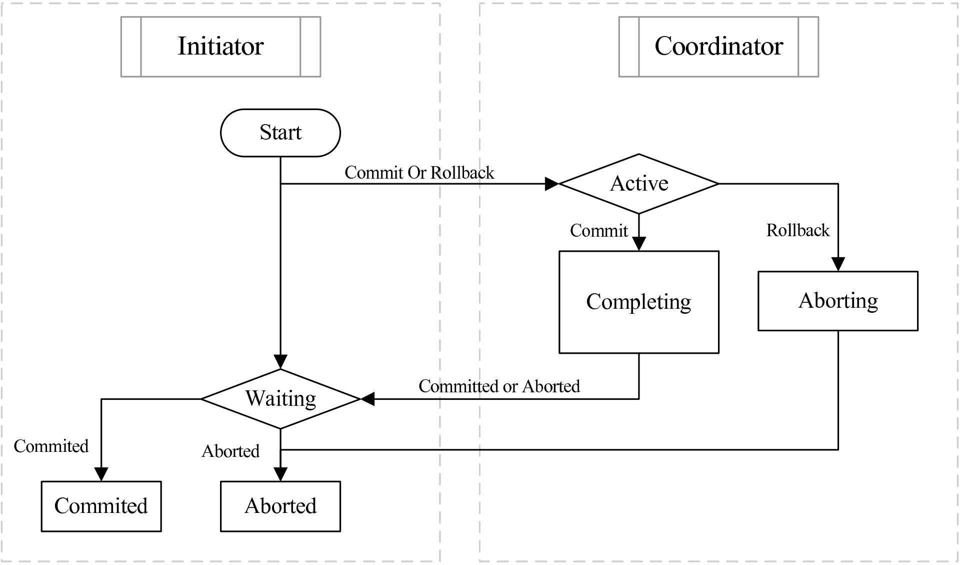

Figure 2: WS-AT Completion Protocol

Two-Phase Commit (2PC) Protocol

The Two-phase commit protocol occurs between the coordinator and the participants. It is used to coordinate registered participants to reach the agreement on the outcome either Commit or Abort the transaction and also to notify all participants accordingly. It contains two phases, the prepare phase and the commit phase. During the first phase, *i.e.* the prepare phase, the coordinator sends a Prepare request to all registered participants soliciting their votes. When the coordinator receives votes from all participants or the request time expires, the second phase, *i.e.* commit phase, begins to notify all the participants with the final outcome of the distributed transaction.

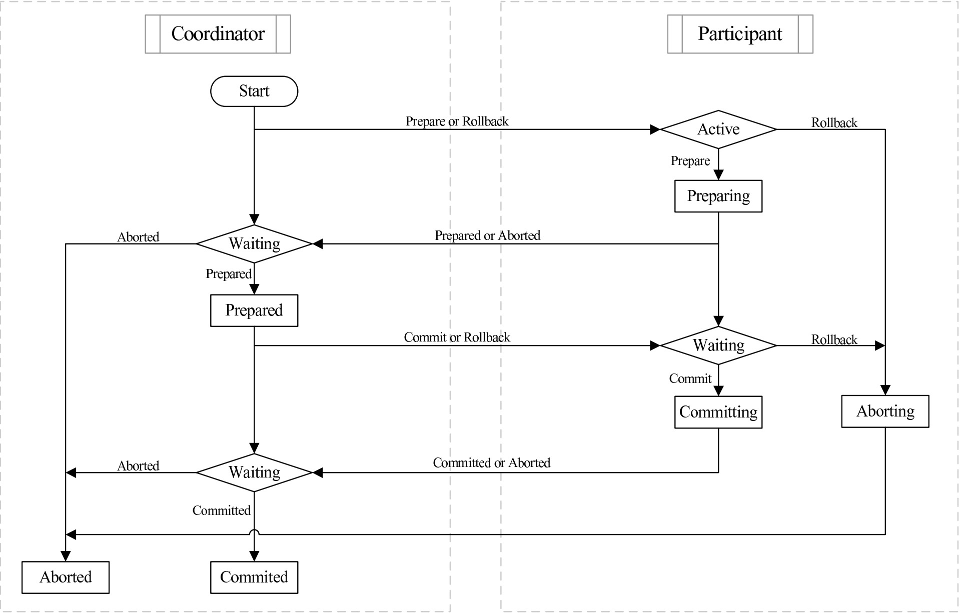

Figure 3: WS-AT 2PC Protocol

2PC protocol has two variants, one is Volatile 2PC used for volatile resources (*i.e.* a cache) and the other one is Durable 2PC, used on durable resources (*i.e.* a database). Participants need to register in the appropriate protocol, possibly more than one, before the transaction starts. A participant registers with several protocols by sending several registration messages. Upon receiving a Commit request from initiator in the completion protocol, the coordinator begins the prepare phase for each participant registered in the Volatile 2PC protocol. Each participant that receives the request must respond before the Coordinator starts the prepare phase in the Durable 2PC. Further participants may continuously register with the Coordinator, but all registrations must be completed prior to beginning of the prepare phase for the Durable 2PC. The coordinator initiates the prepare phase handling for the Durable 2PC protocol by sending out a “Prepare” messages to all participants registered in the Durable 2PC upon completion of the first phase of the Volatile 2PC. All responses must be received before the protocol goes down to the next phase, in which the coordinator will issue the Commit notification to all participants for both the Volatile and the Durable 2PC protocols if all of the feedbacks are positive. If there are any negative votes, even only one, the coordinator has to abort the transaction. After the correct participants receive the Commit notifications, they will commit or abort the transaction and send the acknowledgement back accordingly to the Coordinator.

2.2.2 Web-Service Atomic Transaction Model

Nowadays, the web service is becoming more and more complicated. A web service may be supported by another provided by different companies or organizations. For users, they only need to send a request to the composite Web service through a Web browser (such as Internet Explorer or Firefox Web Browser) or directly invoke the service through application software which stands alone at the client side. To achieve consistent outcome, the one who receives the request should start the distributed transaction to commit interactions. This is the scenario we consider. Figure 4 shows an example of the detailed steps of a distributed transaction supported by WS-AT.

Figure 4 shows a travel reservation example coordinated by WS-Atomic Transaction. In this example, a client contacts a Travel Agent to make travel arrangements on behalf of the client. The Agent, which acts as the completion initiator, is responsible for making reservations for the flight and hotel in the context of an atomic distributed transaction. We assume that the Agent relies on a Flight reservation Web service and a Hotel reservation Web service to book the plane ticket and the hotel room for the traveler. We also assume that these two Web services are managed by different servers in different locations.

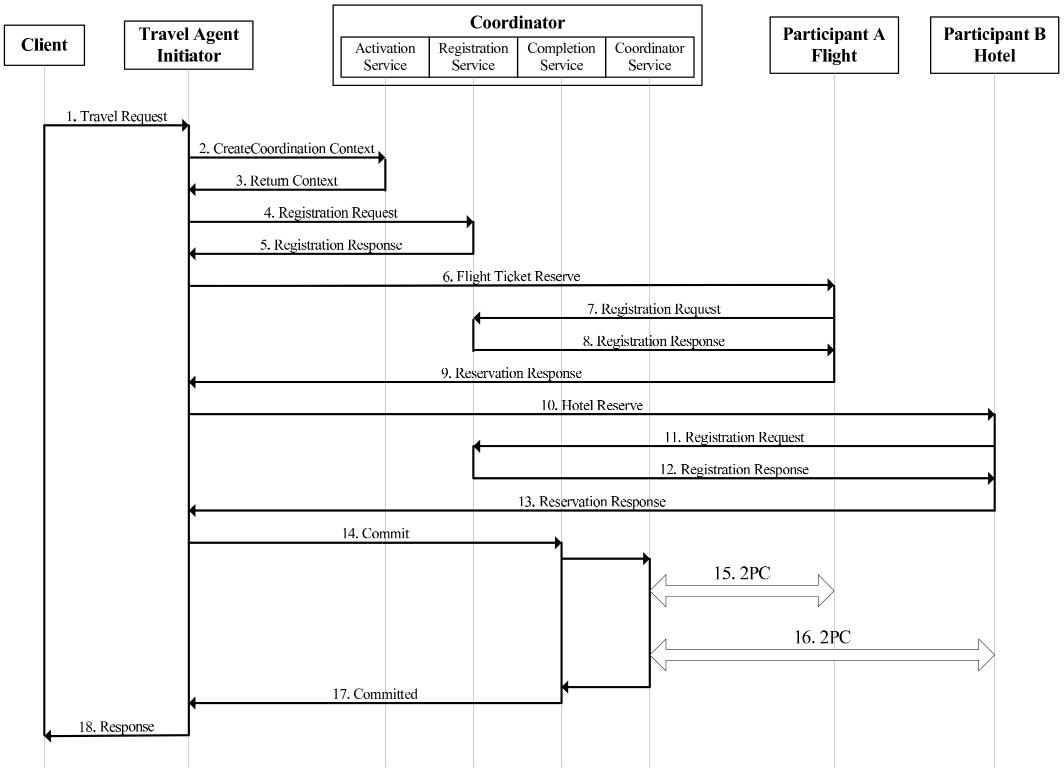

Figure 4: Travel Agent Example Coordinated by WS-AT

To begin this transaction, the client will send a request to the initiator (step 1). The initiator will then invoke the Activation Service on the Coordinator which will create a unique coordination context for the transaction (step 2) containing the Endpoint References of the Registration Service. After the Activation Service, the initiator will receive the transaction context (step 3). In the next step, the initiator will register the Completion Initiator Service with the Coordinator so that the Coordinator could inform the Initiator once it receives the outcome of the transaction (steps 4 and 5). Now the booking service, offered by the initiator, will carry out the reservations for the plane ticket and the hotel room (steps 6 and 10). The flight and hotel booking Web services must register their participant endpoint references with the coordinator (steps 7 & 8 and 11 & 12). After the registration step, these services will send their responses for the reservation requests (steps 9 and 13). Subsequently, the initiator will ask the Completion Service to commit the transaction (steps 14 - 17). Finally, the Travel Agent will send the result back to the client (step 18).

2.3 Byzantine Fault Tolerance

2.3.1 Byzantine Fault

Byzantine, from 5_{th} century to 15_{th} century, is the hoary Roman Empire, which is the city of Constantinople today. In 1453 AD, there was a well known tale that took place within the Byzantine Empire. The city was under siege and there were several powerful Ottoman battalions coped outside an enemy city on different sides poised for the next attack. Each division was commanded by a general, and all commanders would communicate with each other using a messenger service. Because the fortifications of this enemy city were as firm as a rock, no battalion could success by itself. The only way to win this war was to carry out the attack by several, possibly all, of battalions together. Otherwise they should all retreat together. It was apparent that a partial attack would incur heavy losses and infuriate the Grand Sultan, so the generals commanding different camps had to reach the same decision and agree upon the same plan of actions by communicating with other generals. Using the messenger service of the Ottoman Army, the generals were be able share their thinks through messages within an hour. And the receiver was able to certify the identity of the sender and preserve the correct content of the original messages. If all of the generals were loyal, they would be able to reach the same decision fairly quickly. However, some of the generals were treacherous, and they tried to confuse others so that no agreement could be reached. As a result, some of generals would attack while others withdrew. Thus, an insufficient army launched the attack, and unsurprisingly there were heavy losses nearing annihilation for the Byzantine Army.

This is the classic coordination problem now known as the Byzantine Generals Problems. In computer networks, some machines may fall into an arbitrary failure state much like the faulty generals that prohibit others to achieve an agreement. We call this type of failures the Byzantine Faults. The solution to tolerate Byzantine Faults is known as the Byzantine Agreement which describes a way to allow the correct sections to reach an agreement or achieve coordination despite faulty messages.

2.3.2 Byzantine Fault Tolerance

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

The most efficient Byzantine agreement algorithm reported so far was due to Castro and Liskov (referred to as the BFT algorithm). The BFT algorithm is executed by at least $3f + 1$ server replicas to tolerate up to f Byzantine faults. One of the replicas is designated as the primary while the rest are backups. The normal operation of the BFT algorithm involves three phases, normally called the pre-prepare phase, the prepare phase and the commit phase. During the first phase, pre-prepare phase, the primary multicasts a pre-prepare message containing the client's request, the current view and a sequence number assigned to the request to all backups. A backup verifies the request message and the ordering information. If the backup accepts the message, it multicasts a prepare message to all other replicas containing the ordering information and the digest of the ordered requests. This starts the second phase, the prepare phase. A replica waits until it has collected $2f$ matching prepare messages from different replicas before it multicasts a commit message to other replicas, which starts the third phase (commit phase). The commit phase ends when a replica has received $2f+1$ matching commit messages from different replicas. At this point, the request message has been totally ordered and it is ready to be delivered to the server application.

The BFT algorithm is implemented in our Byzantine Fault Tolerant Coordination for Web Services Atomic Transactions system. To avoid the possible confusion between the two phases (the prepare phase and the commit/abort phase) in the Two-Phase Commit (2PC) protocol, we refer the three phases in the BFT algorithm as BA-pre-prepare, BA-prepare, and BA-commit phases in this thesis.

CHAPTER III

BYZANTINE FAULT TOLERANT COORDINATION FOR WEB SERVICES ATOMIC TRANSACTIONS

3.1 System Model

The basic web service system model we considered is a composite Web service as described in section 2.1.1. The end users only need to invoke these services. The distributed transactions, which are used to coordinate the interactions with other Web services, are hidden from the users. The end users will only see the final results when the distributed transactions have been completed.

We use the flat distributed transaction model for simplicity, and implement the WS-AT framework in our system to support the distributed transaction for consistency.

The Service provider, who first accepts the request of the client, acts as the initiator. The responsibilities of the initiator are to start and terminate a transaction, and to also propagate the operations to all other attending participants. The initiator is a special participant, but we will not separate the initiator from the participants unless it is necessary. The only different between the initiator and the participants in our system is that the initiator is replicated to tolerate faults, but for simplicity's sake, we did not replicate participants. Because the initiator is stateless, we only need $2f + 1$ initiator replicas to handle up to f Byzantine faulty initiator replicas. Figure 5 shows the framework of the initiator and the participants (there are only two participants in the figure but can be more).

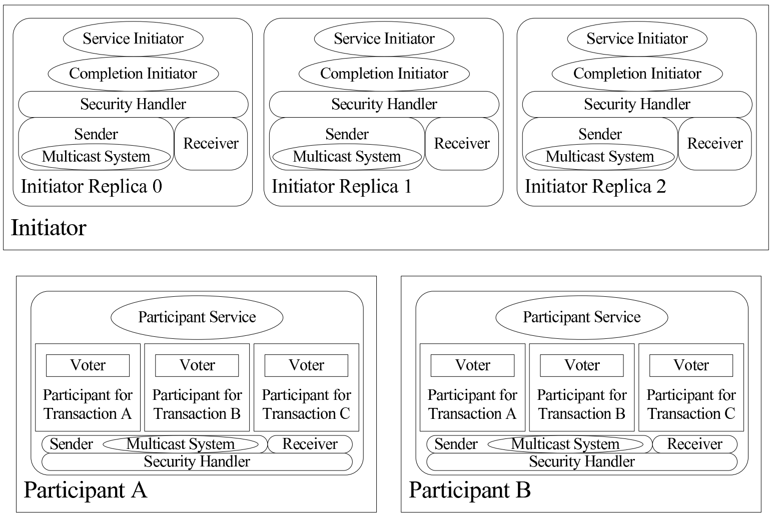

Figure 5: Framework of Initiator and Participants

In our system, the transaction coordinator runs separately from the initiator and the participants. The set of services on the coordinator side runs on the same address space and it is replicated on different machines possibly on many different locations. We use $3f + 1$ coordinator replicas working together to tolerate up to f Byzantine faulty nodes. Figure 6 shows the framework of the coordinator.

This system is a kind of primary-backup mechanism, which means there will always be a primary replica and several backups. To ensure that the system can be still functional when the primary fails, we must pick another as the primary within the replicas in case the current one is faulty. This progress is called view change and

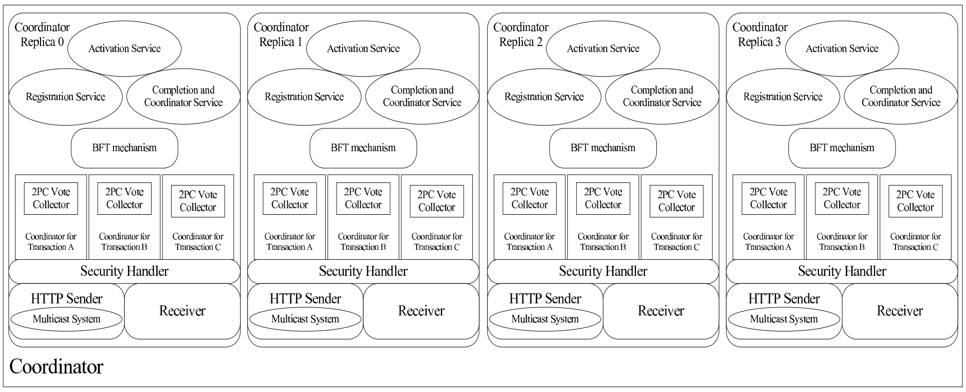

Figure 6: Framework of the Coordinator

we denote a view as v to identify who is the primary. We also allocate a unique ID i, which varies from 0 to $3f$, to every replica. In each view v, one replica, whose ID i satisfies $i = v \; mod \; (3f + 1)$, serves as the primary. If it fails, the next one $i + 1$ will be selected.

At the beginning of each transaction, a coordinator object is created and the lifespan of this object is the same as the corresponding transaction. When the transaction has been completed, the Coordinator of the transaction will cease to exist. During the activation service, we enable an adjusted BFT distributed commit algorithm, which has one more round, to broadcast the collection of message digests. The activation service will be introduced in more detail in section 3.3.

Before each distributed transaction, all correct coordinator replicas need to have the acknowledgments about the registration of the involved participants to ensure that the result of distributed transaction comes from the registered participants set. Furthermore, when the participants receive the message propagated with a registration request from the initiator, they have to register with at least $f + 1$ correct coordinator replicas and then send the reply back to the initiator. If the participant crashes during the registration or before the transaction propagated to itself, either no reply or an exception will be thrown back. Sequentially, the initiator will abort the transaction.

Figure 7 shows the sequence relationship among the main components of the system. The same sequence number means they are taking place at the same time in parallel.

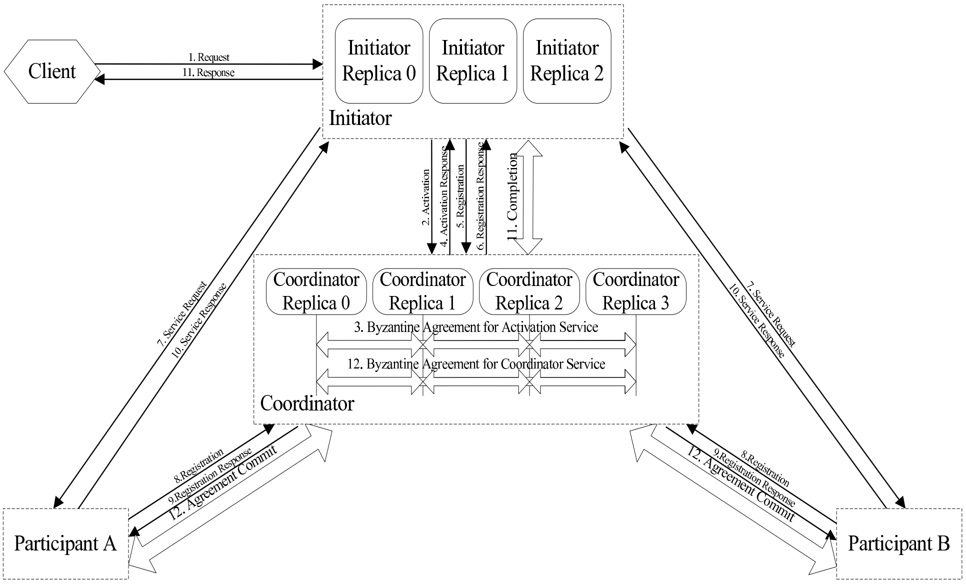

Figure 7: System Framework

We use the Public key algorithm in our system to sign the transaction messages. All messages in the transaction are digitally signed. If confidentiality is needed, messages can be further encrypted. All coordinator replicas and participants have a key pair, a public key and a private key. In this key pair, the private key should be kept secret by the owner and nobody can get it. The public key is well known to others who want to send a signed message to the key owner. I will not go into detail about the digital signature and the encryption algorithms, but we will make the assumption that the encryptions and the digital signatures cannot be broken sufficiently with the current computing power.

3.2 Threat Analysis

Most frequently, all the replicas of the initiator and the coordinator, we believe, should be under the “correct” condition. However, the word “correct” here has different meanings for different actors. We say that the participants are “correct” as long as they are not in the Byzantine Faults. They may, however, fall into the non-malicious faults categories such as a crash or other performance problems. For the coordinator and initiator replicas, we say that they are “correct” only when they faithfully execute all operations and carry out good results accordingly throughout their entire lifetime.

Based on the definition of the “correct” stated above, the coordinator and the initiator replicas are subject to Byzantine Faults which means our system can tolerate arbitrary faulty replicas. On the other hand, for the participants, we have to rule out some forms of the Byzantine faulty behaviors. A Byzantine faulty participant can always vote to commit a transaction but actually abort the transaction locally or vote to abort the transaction but commit it locally. These situations are beyond the scope of any distributed commit protocols. Rather, they should be addressed by business accountability and non-repudiation techniques. Other forms of Byzantine faults on the participants, such as sending conflicting votes to different coordinator replicas, can be tolerated.

Now, let’s take a look at what the faulty parties can impose in order to better tolerate them. Here we enumerate the threats from the Byzantine faulty coordinator replicas and participants.

If coordinator replicas are considered in Byzantine Faults, they can perform some of the following hostile operations:

- Refuse to execute a part of or the whole service request with the intent to block the execution of a distributed transaction. The faulty coordinator replica can do this simply by not responding.

- Abort the transactions despite having received all *yes-votes* from participants. To do this, the faulty coordinator replica omits one or more digitally signed *yes-votes* and pretends those participants did not respond until the timer expires. Please note that the faulty coordinator cannot fake a commit decision if it has not received *yes-votes* from every participant.

- Send conflicting decisions to different participants. The faulty coordinator replicas can send back some correct commit responses and some faked abort decisions after they have already received all *yes-votes* because it is obliged to piggyback all *yes-votes* with a correct commit decision. At the same time, they can fake an abort decision by omitting some votes. The intention is to corrupt data integrity of correct participants.

- Execute the distributed commit protocol correctly for some transactions exactly the same as the correct coordinator.

As a Byzantine faulty participant, it can perform some of the following faulty operations:

- Refuse to execute a part of or the whole distributed commit protocol by not sending or responding, this can cause the involved transactions to abort.

- Vote abort but internally prepare or commit the transaction.

- Vote commit but internally abort the transaction.

As you can see, a faulty participant cannot disrupt the consistency of correct participants as long as the coordinator is correct. To deter malicious participants, the coordinator keeps an auditing log and records all votes from all participants. The logged information can be used to hold a faulty participant accountable for providing fake information. For example, if a participant refused to ship a product that it has promised to, the user and other participants can sue it using the logged vote record from that participant.

Furthermore, for our system, we assume that the coordinator replicas and the participants fail independently. This means that the failed coordinator replicas will not collude with any of the failed participants, this includes the initiator. We do, however, allow failed coordinator replicas to collude amongst each other.

3.3 Byzantine Fault Tolerant Transaction Activation

After analyzing what the faulty parties can impose, let's start how to tolerate them by using our system. Three main parts, BFT Transaction Activation, BFT Registration, and BFT Transaction Completion and Propagation are needed.

Figure 8 shows the details of the BFT Activation service.

The client sends a request to the initiator, which has already been replicated, to start a new transaction. The request is in the form of $<CREQ$, o, t, $c>$ σ_c, where $CREQ$ represents: the request from the client, o represents: the operations client wanted by the client, t represents: the monotonically increasing timestamp, c represents: the id of the client, and σ_c represents: the signature of the client. This request has been sent to all initiator replicas. The initiator replica accepts the request only if the request messages are properly signed by the client and the timestamp t is not smaller than any of the other requests from the same client. After validating that the correct request has been received, the initiator replicas send the activation messages to the primary replica of the activation service to invoke the BFT

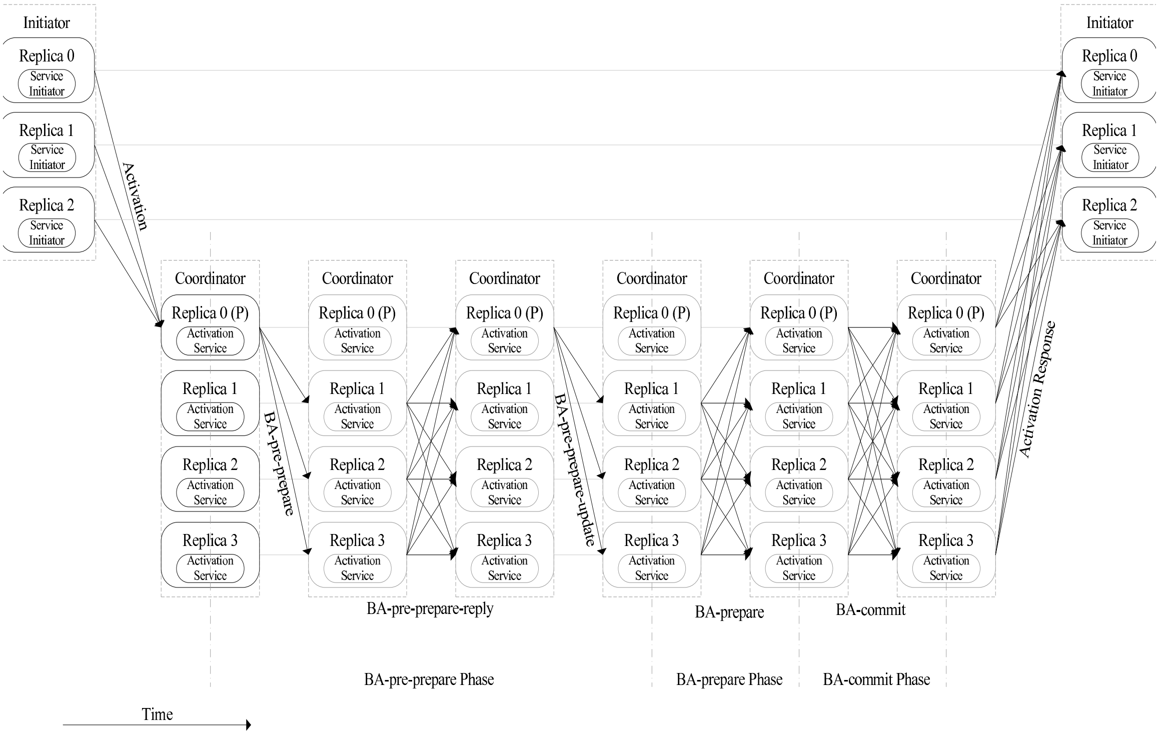

Figure 8: BFT Activation Service

algorithm. The activation request is in the form of $<ACTIVATION$, v, c, t, $k>$ σ_k, where $ACTIVATION$ represents: this is an activation request, v represents: view number, c represents: client id, t represents: timestamp, k represents: initiator replica id, and σ_k represents: The signature of the initiator replica. By using this BFT algorithm, the activation service can handle up to $(T-1)/2$ faults if the service has a total of T replicas (including the primary). Different from the pre-prepare phase of the BFT algorithm, two more transactions (one round between the primary replica and others) are added in. After the primary replicas receive the activation request, it multicast a BA-pre-prepare messages to all other backups. The form of the BA-pre-prepare message is $<$*BA-PRE-PREPARE*, v, r, $uuid_p$, $p>$ σ_p, where *BA-PRE-PREPARE* represents: this is a BA-pre-prepare request, v represents: view number, r represents: contents of the activation request, p represents: the id of the primary replica, $uuid_p$ represents: a universally unique identifier (UUID) proposed by the primary, which is used to generate a identical transaction id to identify the transaction and its coordinator object, and σ_p represents: the signature of the primary replica of the coordinator. For more security [25], the UUID should be generated from a high entropy source. However, this raises a problem. The activation is inherently nondeterministic, since the UUID from one replica cannot be verified by another, this calls for a collective determination of the UUID for the transaction and it is achieved during the BA-pre-prepare phase.

After receiving the BA-pre-prepare message from the primary replica, each backup will verify the authenticity of the message before accepting it. The verification process includes validating the signature of the message, confirming that it is in view v for transaction t and r is a correct activation request. If the message passes verification and the backup has not received the same request before, then the message will be accepted. However, different from the traditional BFT pre-prepare phase, the backup will not go to the prepare phase immediately, instead, it will send a BA-pre-prepare-reply back to all other replicas that contains the digests of the BA-pre-prepare message. This BA-pre-prepare-reply has the form of $<$*BA-PRE-PREPARE-REPLY*, v, d, $uuid_i$, $i$$>$ σ_i, where *BA-PRE-PREPARE-REPLY* represents the type of the message is BA-PRE-PREPARE-REPLY, v represents the current view number, d represents the digest of the BA-pre-prepare message, i represents the replica id, $uuid_i$ represents replica i's proposed universally unique identifier, and σ_i represents the signature of the entire message. The primary accepts the BA-pre-prepare-reply messages if they are properly signed by the sender, they are in the right view for transaction t, and have the correct digests. After the primary gets $2f$ valid replies from different backups, it will send out a BA-pre-prepare-update message in the form of $<$*BA-PRE-PREPARE-UPDATE*, v, d, U, $p$$>$ σ_p, where *BA-PRE-PREPARE-UPDATE* represents the type of the message, v represents the current view number, d represents the digest of the BA-pre-prepare-reply message, U represents the collection of $2f$ digests of the BA-pre-prepare-reply from other $2f$ replicas, p represents the primary id, and σ_p represents the signature of the message. We say a BA-pre-prepare-update message is valid only when the signature is correct, it is in view v, the digest d matches the digest of the BA-pre-prepare-reply message and all digests in collection U are correct.

Upon getting BA-pre-prepare-update message from primary, the progress goes down to the BA-prepare phase and from here the rest of the process are the same as the traditional BFT algorithm except that the focus is placed on the agreement upon the outcome instead of the total ordering. The BA-prepare message has the form of $<BA\text{-}PREPARE, v, d, uuid, i> \sigma_i$, where v represents the current view number, d represents the digest of the BA-pre-prepare message, i represents the replica id, and σ_i represents the signature of the message. This uuid is the final UUID calculated based on $2f$ backups' proposed UUID. There may be multiple computation methods but we use a very simple one which just calculate the average as the final UUID. The replica will check all parameters in the message to verify its correctness.

When the replica (including the primary) has received $2f$ valid BA-prepare messages from the different replicas (including the one from itself), it will multicast a BA-commit message out in the form $<BA\text{-}COMMIT, v, d, uuid, i> \sigma_i$, where $BA\text{-}COMMIT$ represents the message type is BA-commit, v represents the current view number, d represents the digest of the BA-prepare message, uuid represents the universally unique identifier, i represents the replica id, and σ_i represents the signature of this BA-commit message. Like every step before, the receiver must validate message first. The validation for a BA-commit message should include view number v, digest d, final UUID and the signature.

When a replica gets $2f+1$ matching BA-commit messages from different replicas (including the one from itself), it computes the transaction identification which we call tid by using the uuid and creates a new coordinator object for the transaction request with the tid, then sends back the activation response to all replicas of the initiator. The response message has the form <$ACTIVATION$-$RESPONSE$, c, t, C, i> σ_i, where c represents the client id, t represents the timestamp, C represents the context of the transaction, i represents the replica id number, and σ_i represents the signature of message.

The initiator replica logs the activation response after verification. The message will be accepted if $f + 1$ matching messages from different activation service replicas are received.

3.4 Byzantine Fault Tolerant Registration

Each participant has to go through the registration process so that the coordinator replicas have the acknowledgment about who is involved in the transaction to ensure that the atomic transaction can be executed correctly. No doubt running a Byzantine agreement algorithm for each registering participant can help coordinator replicas to reach an agreement. But it is too costly to be practical. So we defer the Byzantine agreement to the distributed commit stage on the participants set and combine it with that for the consistent transaction outcome. Figure 9 shows the details of the registration service.

The registration service will be done the normal way. Please notice that, because we have several initiator replicas, the participant only accepts the request after it gets at least $f + 1$ matching messages. This prevents the progress from the faulty initiator replica because we will have at least 1 correct request from the loyal initiator backup since we only have at most f faulty replicas.

To register, a participant sends the request to all replicas of the registration

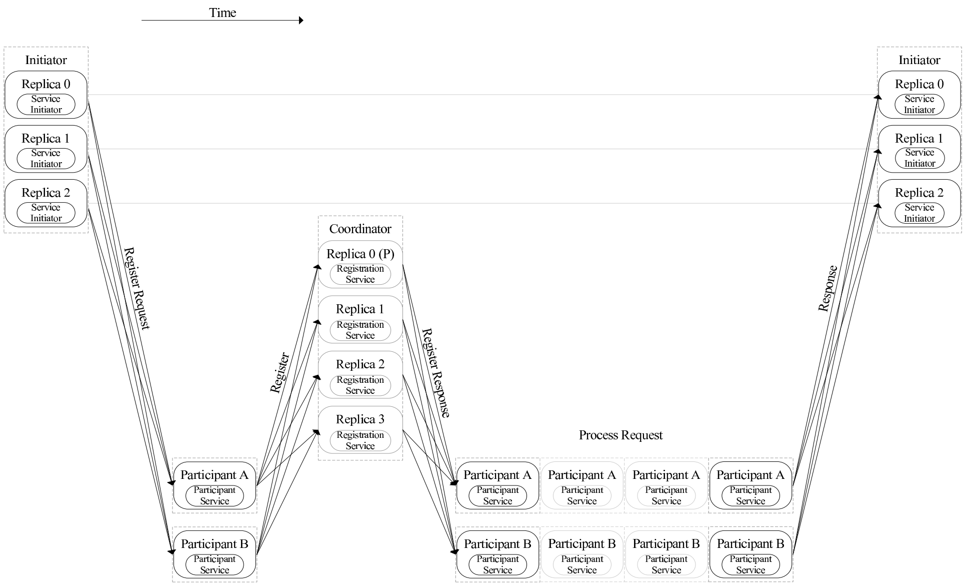

Figure 9: BFT Registration Service

service and accepts the reply from them not earlier than $2f+1$ acknowledgments back. We assume that there are at most f faulty replicas. To tolerate f faulty ones, we need at least $f+1$ correct replicas to accept the registration. After successful registration, participant will execute the initiator's request and send a normal response to the initiator replicas. If the initiator replicas get an exception or no response from the participant at all, the transaction will be aborted. The initiator, as a participant, has to do the registration as well.

3.5 Byzantine Fault Tolerant Transaction Completion and Propagation

Comparing against the 2PC protocol, we have two main differences. The first one is that we have an additional round on the coordinator side in order to reach an agreement on the transaction outcome. The second is that the decision from the coordinator replica to the participants is queued until we can get at least $f+1$

identical decision messages. Because we assume that there are f faulty coordinator replicas, if we can reach the same decision messages from more than f nodes, parts or all of these decision messages must come from the loyal coordinator replicas.

The initiator, as in the normal atomic transactions, is responsible for starting and terminating a distributed commit for a transaction. When the initiator has completed all operations within a transaction, it will send the commit or rollback request to a coordinator replica based on whether the operations were successful or not. The coordinator replica will accept the request after it has received at least $f+1$ matching requests from the different initiator replicas. Upon receiving the commit request, the primary will invoke the Byzantine Fault Tolerance Algorithm starting from the first phase of the 2PC. However, if the request is to rollback, this phase of the 2PC is skipped. For all requests, other steps next to the first phase are needed to reach the agreement on the outcome and inform all initiator replicas of final results.

The completion service and propagation are shown in Figure 10. Figure 11 shows the details about the Byzantine agreement algorithm used in the completion service, which contains three phases as described before.

The first phase of the BFT distributed commit protocol is the Prepare phase of the 2PC. During this phase, the coordinator replicas send a Prepare request to every participant registered with the coordinator. In this request, the coordinator contains the request which was sent from and also signed by the initiator. When the participant gets the request, it starts the signature verification. Furthermore, if the participant knows the public key of the initiator, he will continuously check the signature of the original request. If not, this step is ignored. Based on the result of the verification, the participant makes the decision to accept the request or not. If there are any problems during the verification, the request will be discarded. This can help us protect the participants from faulty requests, which may come from the

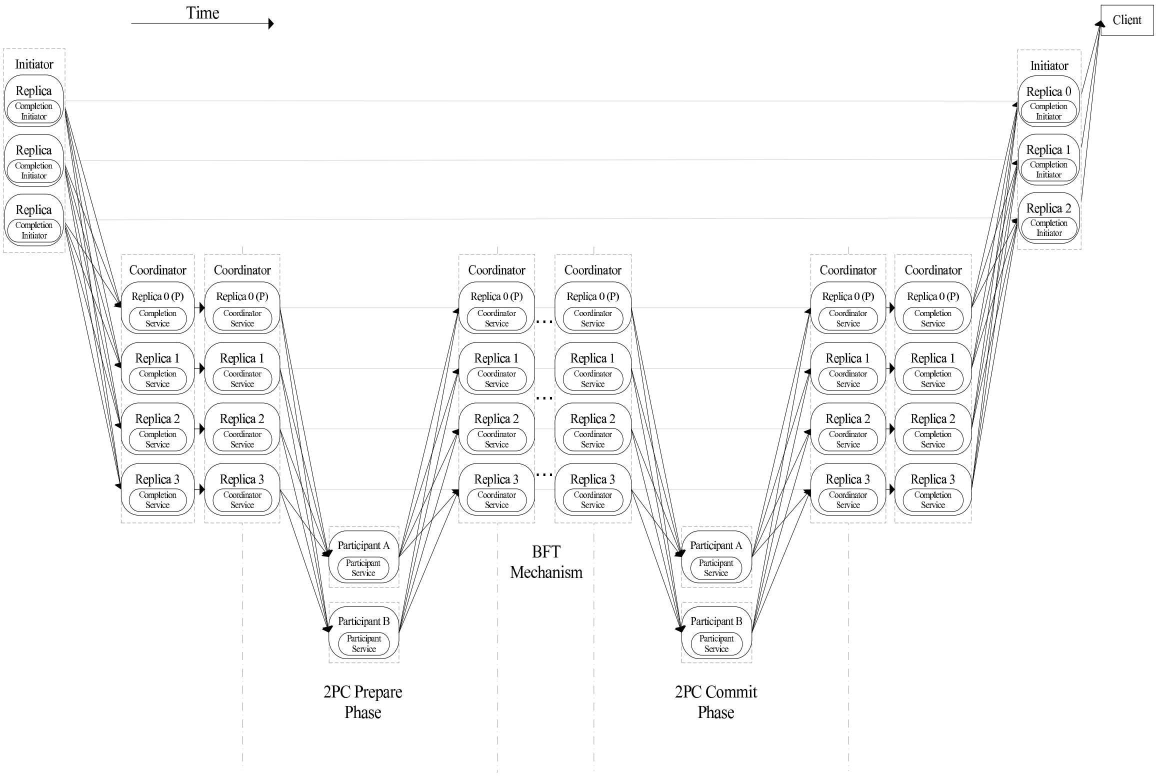

Figure 10: BFT Completion and Coordination Services (1) 2PC

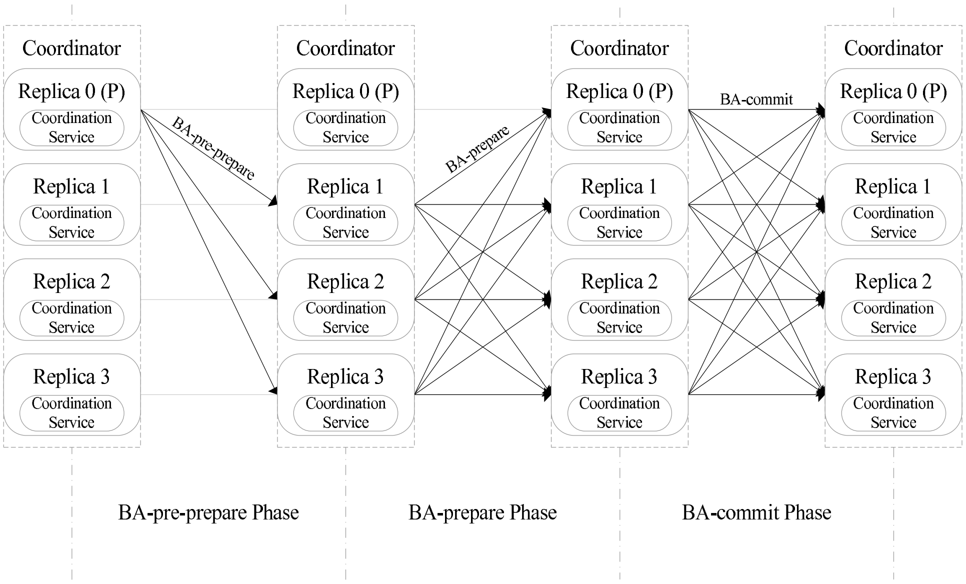

Figure 11: BFT Completion and Coordination Services (2) BFT Mechanisms

faulty coordinator replicas, because the faulty coordinator replicas may try to confuse participants by sending the different requests to them.

After the prepare phase, all correct coordinator replicas begin the Byzantine agreement algorithm to reach the consistent outcome. There are three phases in the Byzantine agreement algorithm, just like the BFT algorithm for activation service, which are the BA-pre-prepare phase, the BA-prepare phase and the BA-commit phase. The algorithm starts with BA-pre-prepare phase. The primary coordinator replica sends his decision, which contains a set of records from all participants, to all other replicas. The BA-pre-prepare message is in the form of $<BA\text{-}PRE\text{-}PREPARE, v, t, o, C> \sigma_p$, in which *BA-PRE-PREPARE* represents it is the BA-pre-prepare message, v represents the view number, t represents the transaction id, o represents the sender's decision about the outcome, C represents the decision certification, and σ_p represents the signature of the primary. The unique transaction id is used to avoid the faulty replica reusing the message. When other replicas get the BA-pre-prepare message, they will first verify the following four parts:

−Verify the signature σ_p. And check the view number v and the transaction id t.

−Verify it is the first time the BA-pre-prepare message was received in view v for transaction t.

−Verify that all records in C are properly signed, the transaction id matches the current transaction, and the decision o is identical to the registration and vote results.

Note that a backup does not insist on receiving a decision certificate identical to its local copy. This is because a correct primary might have received a registration from a participant which the backup has not, or the primary and backups might have received different votes from some Byzantine faulty participants, or the primary might have received a vote that a backup has not received if the sending participant crashed right after it has sent its vote to the primary.

If the registration records in C are different from the local records, the replica, as the receiver, has to update its registration list by asking the missing information from the primary.

If the replica accepts the message, it logs the accepted message and is ready to go to the BA-prepare phase. This state is called the BA-pre-prepared state. Otherwise, if the verification fails, the message will be discarded and the replica will suspect this primary because it believes that the current primary is faulty from its point of view. He has to invoke a view change to select a new primary instead of the faulty one.

After the BA-pre-prepare phase, the replica gets a proposed decision from the primary. It goes to the BA-prepare phase and multicast the BA-prepare message to all other replicas including the primary. The message is in the form $<BA\text{-}PREPARE$, v, t, d, o, $i>$ σ_i, where $BA\text{-}PREPARE$ represents the type of the message, v represents the view number, t represents the transaction number, d represents the digest of the decision certification C, o represents the sender's proposed decision, i represents the replica's id and σ_i represents the signature of the message.

The receiver will check the prepare message using the following steps:

−Check the signature, view number and the transaction id first.

−Check that whether the proposed decision o is the same as the one received from the BA-pre-prepare message.

−Check that whether the digest d matches the decision certification digest in the BA-pre-prepare message.

After the replica collects $2f$ accepted BA-prepare messages from the different replicas, it can make the decision for transaction t, which means this replica has reached the BA-prepared state.

A BA-prepared replica will continue to the BA-commit phase by multicasting the BA-commit message, which is in the form of $<BA\text{-}COMMIT$, v, t, d, o, $i>$ σ_i. v, t, d, o, i and σ_i have the same meaning as in the BA-prepare message. So when the coordinator replicas get the BA-commit request, the same verification methods for BA-prepare messages will be used to ensure validity. If the coordinator replica i could get $2f+1$ matching BA-commit messages from different backups (including the one from itself), then we know that at least $f+1$ correct coordinator replicas will reply with the consistent final decision to the participants, since there are at most f faulty ones. Now the coordinator replica has reached the BA-committed state and the Byzantine agreement algorithm has been completed. In the next step, this replica will send the final decision to all registered participants through the Commit phase. When the coordinator replicas receive the Committed replies from all participants, it will notify the initiator that the distributed commit progress has been completed. Then the initiator replicas will send this reply to the client.

However if one replica cannot reach the BA-committed state until timeout or it gets an invalid message from the primary, this replica has to invoke a view change by sending the view change request to all replicas. The view change message is in the form of $<VIEW\text{-}CHANGE$, $v+1$, t, P, $i>$ σ_i, where $VIEW\text{-}CHANGE$ represents the quest type, $v+1$ represents the preferred new view number, t represents the transaction id, and P represents the current state of the replica i. If the replica i reaches different states, then P contains different information. For BA-pre-prepared, P includes $<v$, t, o, $C>$. If the replica has reached BA-prepared state, P includes $<v$, t, o, $C>$ and $2f$ matching BA-prepared messages from the different replicas. If state of replica i is still before BA-pre-prepared for transaction t, the current state information only contains his own decision certificate C.

When the correct replica receives the view change request from other replicas, the current view will not be changed immediately, instead, it will wait for at least

$f + 1$ view change requests from different replicas before making the decision to multicast its view change request to all other replicas for view $v+1$ to protect against unnecessary view changes.

After the new primary gets at least $2f + 1$ valid view change requests for view $v + 1$ (including the one from itself), it takes care of the new view and notifies all other replicas with the new view message in the form of $<$*NEW-VIEW*, $v + 1$, V, t, o, $C>$. In this new view notification, V is a collection of $2f + 1$ tuples for new view $v + 1$ from other replicas. Each tuple has two parameters $<i$, $d>$ which means the view change request is from replica i and the request digest is d. The o and C in the new view notification have the same meaning as before, o is the decision and C is the decision certificate. These two parameters have different contents depends on the following rules. If the view change message includes a valid BA-prepare message and all records are consistent, then the primary must fail after successfully completing first two phases of the BFT distributed commit protocol. The decision will be accepted by the new primary and the whole transaction will move down to the BA-commit phase. Otherwise, the new primary rebuilds a set of registration records from the received view change messages. This new set may be identical to, or be a superset of, the registration set known to the new primary prior to the view change. The new primary then rebuilds a set of vote records in a similar manner. It is possible that conflicting vote records are found from the same participant (*i.e.*, a participant sent a “prepared” vote to one coordinator replica, while it sent an “aborted” vote to some other replicas), in which case, a decision has to be made on the direction of the transaction t. In this work, we choose to take the “prepared” vote to maximize the commit rate. A new decision certificate will be constructed and a decision for t’s outcome is proposed accordingly. They will be included in the new view message for view $v + 1$.

If a replica gets a new view change message, validation with the same steps used by the primary, is still needed. If a replica accepts the view change, it has to update the information about the endpoint references for the participants. Then the progress will continue to work as usual.

CHAPTER IV

IMPLEMENTATION AND PERFORMANCE EVALUATION

Our Byzantine Fault Tolerant WS-AT Coordination framework is built on top of the Kandula project from apache.org. The Kandula project is an open-source implementation of Web Service Coordination Specification [17] and Web Service Atomic Transaction Specifications [10] (we introduced in section 2.2.1) based on Axis in the Java programming language. Our framework also used some other Apache Web services projects, including WSS4J (an implementation of the Web Services Security Specification) [5], and Apache Axis (the SOAP Engine) [3]. Most of the BFT WS-AT Coordination mechanisms are implemented in terms of Axis handlers that can be plugged into the framework without affecting other components. Parts of Kandula code is modified to enable higher level control of its internal state and to enable the Byzantine agreement algorithm and voting mechanism. In this section, we first introduce the technical details about the framework implementation including Multicaster, Piggybacking, Voter and Vote Collector, Byzantine Agreement Agents, and Security handler.

4.1 Byzantine Fault Tolerant Coordination Framework for WS-AT

4.1.1 Multicaster

To enable message exchanges among the replicas and between the clients and the replicas, multicaster is needed. The Multicaster is responsible to ensure reliable multicast of a message to a group of receivers. This Multicaster carries out the multicast by using multiple Point-to-Point messages on top of the SOAP protocol for maximum interoperability.

On the sending side, a thread pool is used to concurrently send the multiple messages to their destinations to achieve good performance, because the HTTP server synchronously waits for the response after it sends out a request. The thread pool is used here so that each time a message has been sent out to a target it can be handled by a different thread. In this case, the HTTP server in a different thread can wait there until a reply has been received or the timer has timed out. We named the two parts of the Multicaster in our framework as MyHTTPSender, which is used to execute the high level processing such as creating a thread pool and finding the address of the destination, and MyHTTPSenderWorker, which is responsible for sending the SOAP message out and waiting for the reply. The flowchart of the Multicaster is shown in Figure 12.

Again, the Multicaster is used to send the same messages to all replicas of initiator or coordinator. After getting a message to be sent by the Multicaster, the thread pool is created first. Then based on different requests, the destination addresses for the replicas are found and the targets' URLs are assembled. Finally, a

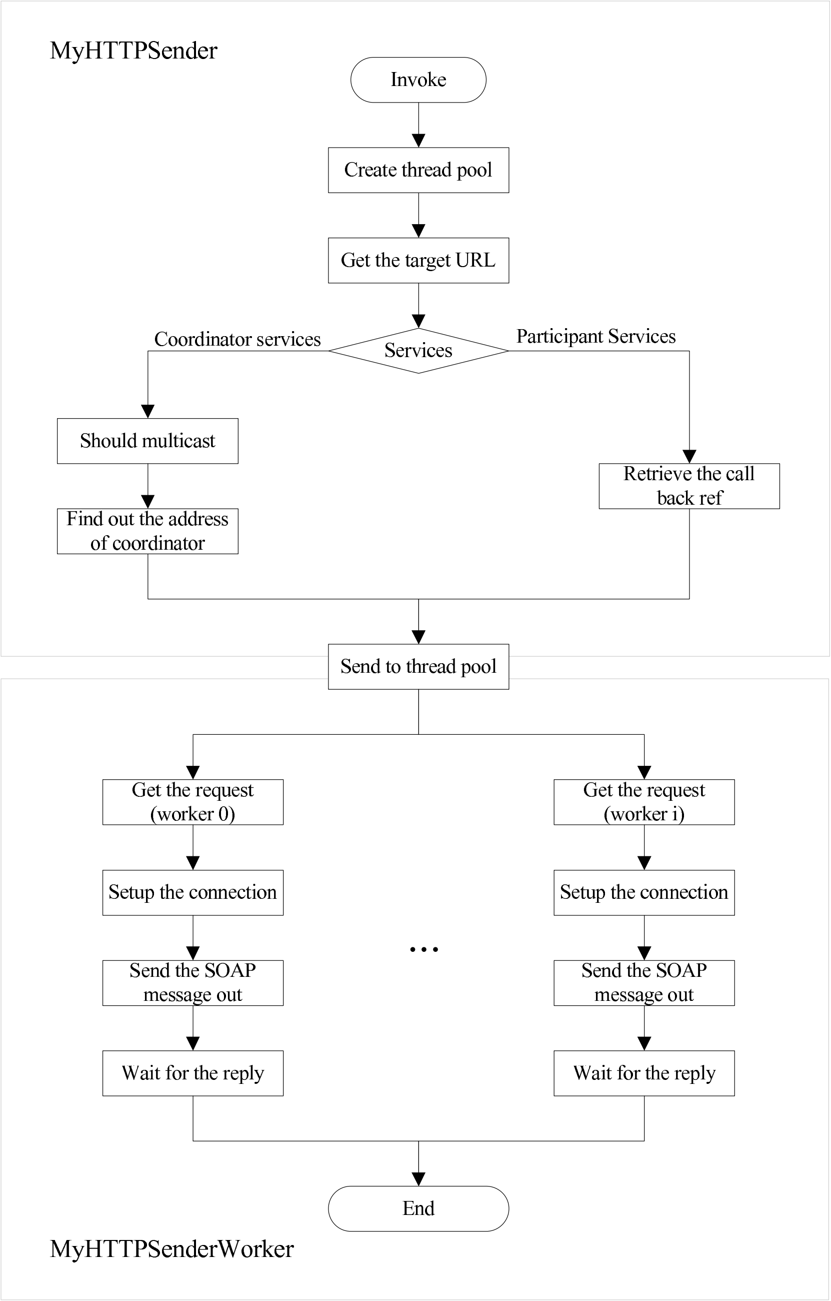

Figure 12: Multicaster Flowchart

thread pool is created to send out these messages separately.

For simplicity, our implementation of the reliable Multicaster will use a static membership provided by a configuration file.

4.1.2 Piggybacking

In the 2PC protocol, the coordinator might send three different requests to the participants: prepare, commit and abort. The participants will vote by prepared, committed and aborted based on their decisions. From the beginning of the Byzantine agreement algorithm, all votes have to be piggybacked onto the message header to restrict what a faulty coordinator replica can do to compromise the atomicity. A similar piggybacking idea is first mentioned in [4].

In our framework, the piggyback is realized by two main function methods, *i.e.*, PiggybackVoteWithDecision() and VerifyDecisionWithPiggybackedVotes(). The method PiggybackVoteWithDecision() is used to insert the vote records into the SOAP message header, and VerifyDecisionWithPiggybackedVotes() is responsible for extracting the piggybacked vote records from the SOAP message and returning a vector containing all reconstructed vote messages for further progress if it goes well. If there are any problems occur during verification, the returned vector will be null. Figure 13 shows the details about the piggybacking.

The immediate benefit of using piggybacking is to prevent a faulty coordinator replica from sending conflicting decision messages to different participants without being detected, (*i.e.*, if some participants voted to abort the transaction, or indeed has failed or did not respond). This is because a commit decision message must piggyback with a token containing a complete set of yes-vote and a faulty coordinator replica cannot find a way to fabricate a yes-vote without knowing the private key of the corresponding participant. This is true as long as the faulty coordinator does not

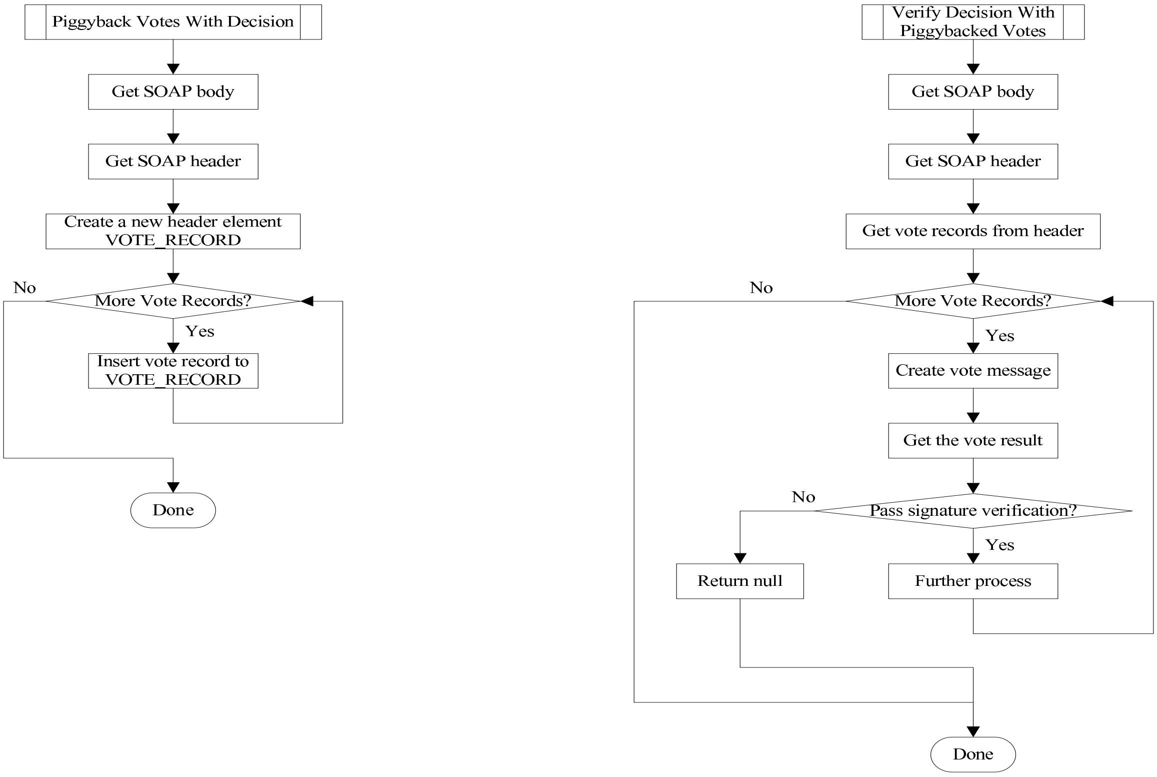
Piggyback Votes With Decision
Get SOAP body
Get SOAP header
Create a new header element VOTE_RECORD
No
More Vote Records?
Yes
Insert vote record to VOTE_RECORD
Done
Verify Decision With Piggybacked Votes
Get SOAP body
Get SOAP header
Get vote records from header
No
More Vote Records?
Yes
Create vote message
Get the vote result
No
Pass signature verification?
Yes
Return null
Further process
Done

Figure 13: Piggybacking Flowchart

collude with any of the faulty participants, which was our assumption.

Therefore, because of the power limitation of the piggybacking, a faulty coordinator replica can still disseminate conflicting decisions to the participants only when all participants have voted to commit a transaction. There are only two “legitimate” ways to do so:

1. The faulty replica can send a commit decision to some participants, but an abort decision to others by falsely claiming that it did not receive the vote from one or more participants. In fact, the faulty replica could send the abort decision to a subset of participants as soon as the distributed commit starts without going through the first phase.

2. The faulty replica can send a commit decision to some participants, but nothing at all to some other participants, hoping that the subset of participants that do

not receive a decision to indefinitely hold valuable resources for the transaction, or the participants to unilaterally abort the transaction due to a timeout.

Therefore, to tolerate the faulty coordinator and initiator replicas and participants and to obtain consistent final outcomes, the Voter & Vote Collector and the Byzantine Agreement Agents are needed, which will be introduced in next two sections.

4.1.3 Voter and Vote Collector

In our framework, we use 2PC, which is a kind of voting mechanism [24], to commit the outcome and to notify participants of the final decision. This voting mechanism is realized through two parts: 2PC Voter and 2PC Vote Collector. For each transaction, one voter object for each participant and one vote collector object for every coordinator replica are created. Both voter and vote collector exist for a single transaction, and will be destroyed when the transaction is completed. The lifespan of the voter and vote collector objects are identical to the coordinator object.

The 2PC starts when the coordinator replicas accept the commit request from the completion initiator. The coordinator replicas send a prepare request to all participants piggybacked with the proposed outcome. The participant votes the transaction by using the 2PC voter if the request is valid. The voter will compare the proposed outcome with the records and reply the coordinator with either prepared, which means the outcome is correct, or abort, which means the outcome mismatch with the records.

The 2PC Vote Collector is responsible for getting and storing the digitally signed vote messages from the participants' voter. Later these vote messages will be piggybacked with further messages for the Byzantine Agreement, which will be introduced in the next section.

4.1.4 Byzantine Agreement Agents

The main focus of our framework is to tolerate the Byzantine Faulty Coordinator and Initiator replicas by using the Byzantine Agreement Agents. In our framework, there are two different Byzantine Agreement Agents, one for Activation service and the other one for Coordinator & Completion services, to help us achieve the goal. Three parts, BA-Sender, BA-Executer and BA-handler, are needed in our Byzantine Agreement Agents to send, execute and handle the SOAP messages. The main difference between the two agents for activation service and coordinator & completion services is that one more round is added in the BA-pre-prepare phase for the activation service to collect the digests from other coordinator replicas. We call these two messages BA-pre-prepare-reply and BA-pre-prepare-update.

Byzantine Agreement Agent for Activation Service

The basic idea of the activation service was provided in section 3.3. To implement it, those three parts are needed, BA-Sender, BA-Executer and BA-handler. Each of them has different responsibilities. A BA-Sender is used to send the messages out, such as BA-pre-prepare, BA-pre-prepare-update, BA-prepare and BA-commit. We wrote a sender for each of these four kinds of messages and we call all of these four senders together as BA-Sender for simplicity. Two of these four, BA- pre-prepare and BA-pre-prepare-update, are from the primary replica. So only the primary will invoke sendBapreprepare() and sendBapreprepareupdate(). The first step of BA here is to check whether the replica itself is current primary or not, which can be done by using a method named amPrimary() in the config.java file. Because we only enable the exception of the view change mechanism in our implementation, we did not write the code that handles view change. In this case, we pick the first replica, whose id is 0, as the primary. Accordingly, the first replica will get true from the feedback of the amPrimary() method. At the beginning of sendBapreprepare(), we are not in a hurry to send out the messages. Instead, we need to judge whether another BA is already in the process or not. If so, we have to queue the current BA request until all previous BAs are finished. Otherwise the BA will start by sending out BA-pre-prepare messages to all other replicas. As the normal backup (except the primary), they start the Byzantine Agreement operation directly from the BA-pre-prepare Executer which is named as doBaPrePrepare(). We have to separate this executer into two parts by judging whether it is primary again, because this executer is used by both primary and backups to handle the BA-pre-prepare phase. As the primary replica, it is responsible for collecting the BA-pre-prepare-reply and then sending out the BA-pre-prepare-update to other backups if it gets $2f$ replies. On the other hand, the backups use the doBaPrePrepare() method to take care of both pre-prepare and pre-prepare-update messages. In doBaPrePrepare(), we invoke another class called BACertificate() which contains two functions add() and isComplete(). We justify whether the current progress is completed by calling isComplete(), which will return true if the valid messages are more than the threshold. Base on the feedback of the certificate, the progress will either wait for the incoming messages if it is not complete yet or move down to the next phase if the current phase is done. The BA-pre-prepare handler (handleBaPrePrepare()) is responsible for handling every new arriving message. It uses another function call in BACertificate add() to increase the counter of the message. Then justify the progress again. If it gets enough messages in the collection, the doBaPrePrepare() will be waked up from waiting by using notifyAll(). Figure 14 shows the flowchart of the BA-pre-prepare phase for activation service.

In next two phases of the Byzantine agreement, the same three components BA-Sender, BA-Executer and BA-handler are involved. However, they are not as complicated as in the BA-pre-prepare phase. There is no need to justify who is the

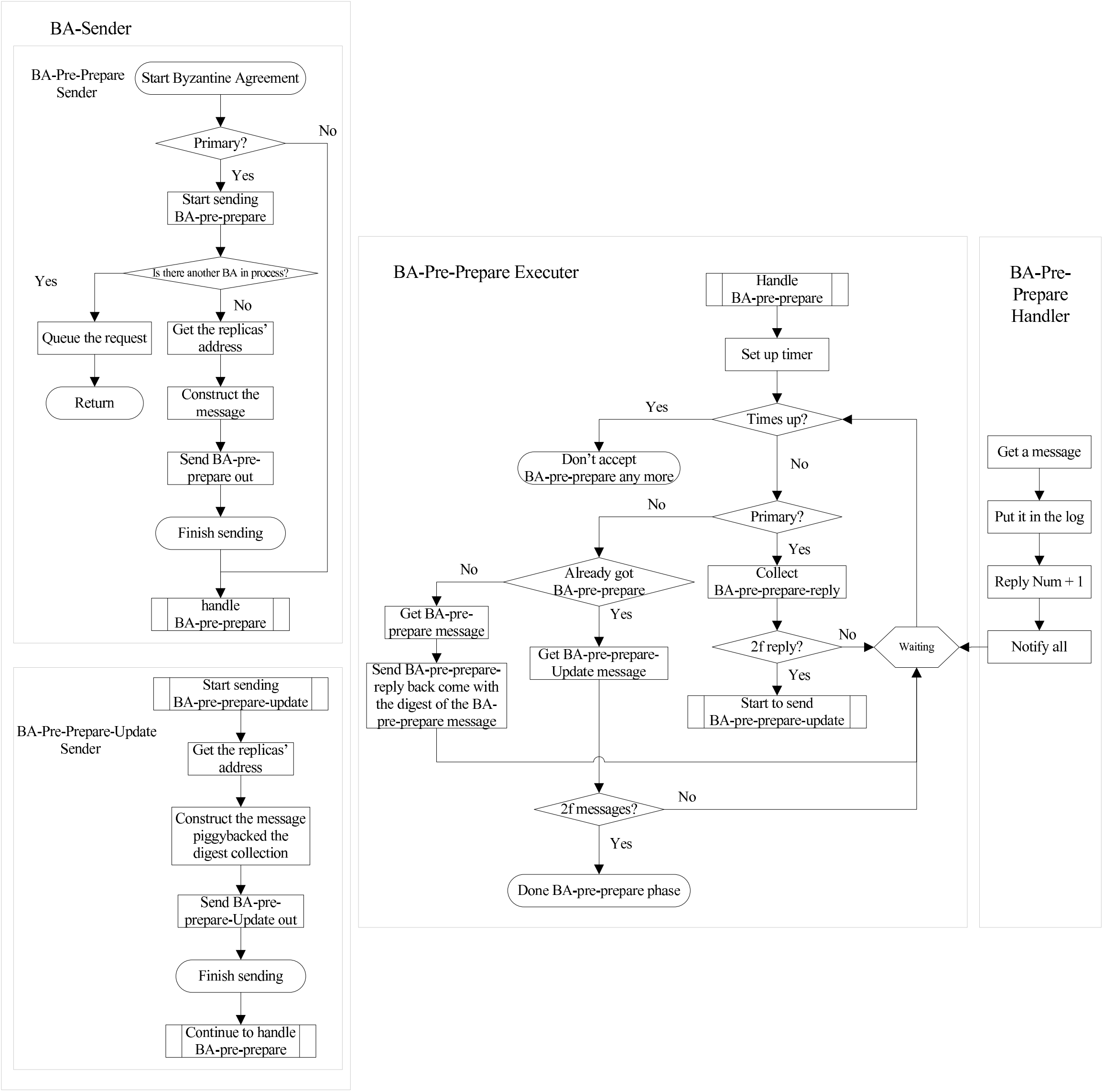
BA-Sender
BA-Pre-Prepare Sender
Start Byzantine Agreement
Primary?
No
Yes
Start sending BA-pre-prepare
Is there another BA in process?
Yes
No
Queue the request
Get the replicas' address
Return
Construct the message
Send BA-pre-prepare out
Finish sending
handle BA-pre-prepare
Start sending BA-pre-prepare-update
BA-Pre-Prepare-Update Sender
Get the replicas' address
Construct the message piggybacked the digest collection
Send BA-pre-prepare-Update out
Finish sending
Continue to handle BA-pre-prepare
BA-Pre-Prepare Executer
Handle BA-pre-prepare
Set up timer
Yes
Times up?
Don't accept BA-pre-prepare any more
No
No
Primary?
Yes
No
Already got BA-pre-prepare
Collect BA-pre-prepare-reply
Yes
Get BA-pre-prepare message
Get BA-pre-prepare-Update message
2f reply?
No
Waiting
Send BA-pre-prepare-reply back come with the digest of the BA-pre-prepare message
Yes
Start to send BA-pre-prepare-update
2f messages?
No
Yes
Done BA-pre-prepare phase
BA-Pre-Prepare Handler
Get a message
Put it in the log
Reply Num + 1
Notify all

Figure 14: Flowchart of BA-Pre-Prepare phase for Activation Service

primary anymore because the primary and the other replicas will perform the same operations during the next two phases. The BA-Senders for both of the prepare phase and the commit phase are responsible for sending the BA-prepare or BA-commit messages out. BA-Executers set up the timer and make the decision to either finish that phase or wait for more messages. BA-Handlers are used to take care of every new message and wake up the Executer from the waiting. Figure 15 and 16 are the flowcharts of these two phases for the Activation service. When we finish the Byzantine agreement, we need to remove its id from the queue and try to start the next one if there are more.

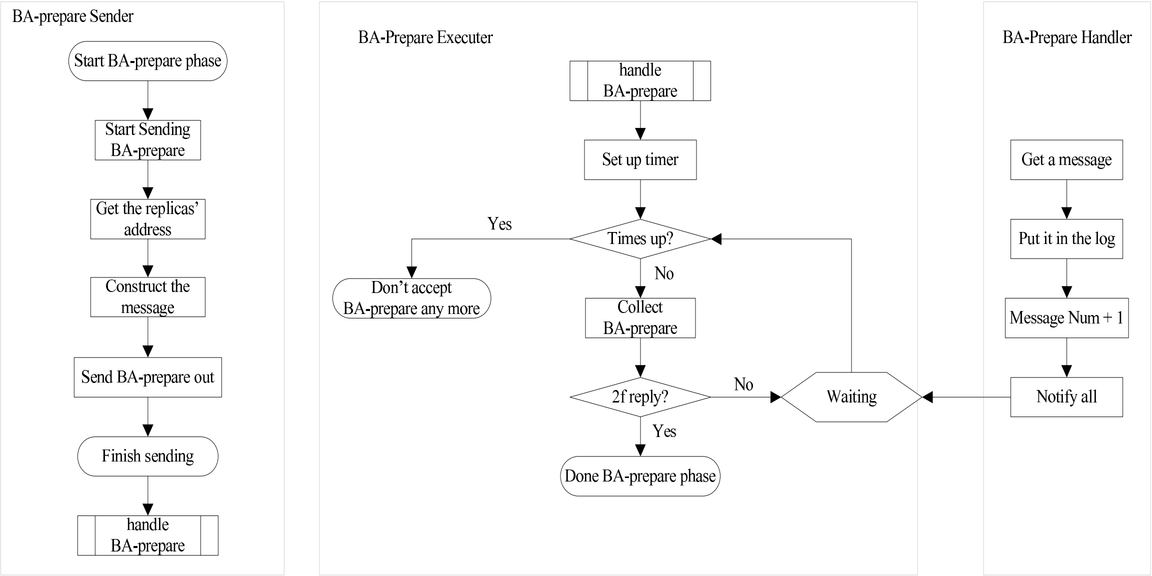

Figure 15: Flowchart of BA-prepare phase for Activation Service

The Byzantine Agreement algorithm for Coordinator and Completion Services is similar to BA for Activation Service, except there is no additional round in the first phase. Therefore the pre-prepare phase is much simpler than the previous one. Different from the BA-pre-prepare phase for Activation service, we only use two parts to implement BA-pre-prepare phase in this Byzantine Agreement Agent, BA-Sender and BA-Handler, because we don't need to collect the number of messages. Every backup will get one BA-pre-prepare message if there is no problem during the

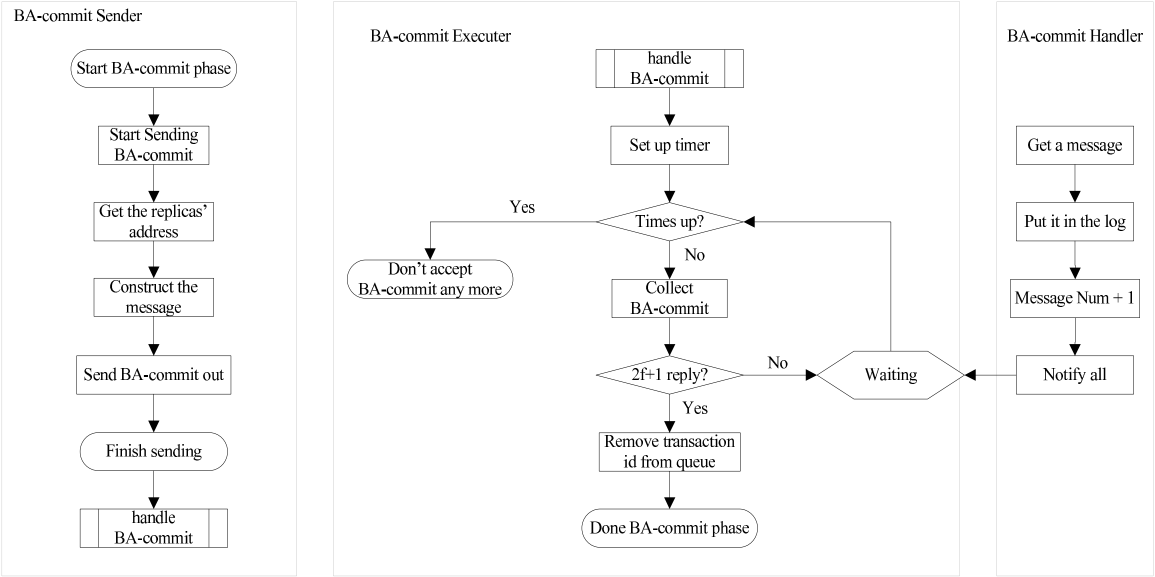

Figure 16: Flowchart of BA-commit phase for Activation Service

transaction. In the BA-pre-prepare phase, the primary only needs to send out the BA-pre-prepare messages to all other replicas via BA-sender. As the backups, when they get the BA-pre-prepare message, the BA-handler is used to get the information it wants. Figure 17 shows the flowchart of the BA-pre-prepare phase.

In the BA-prepare and BA-commit phases, we also use the same three parts as in BA for activation service. The BA sender is very simple in both the BA-prepare phase and the BA-commit phase. The only task of the sender is to find the address of the destination, construct the message, and then send the message out. Then the BA-Executer will be used to set up the timer and wait for the notification from the Handler. If the BA-Handler gets a message and it is still in progress, the handler will increase the counter by 1 and check whether it has reached the minimum limitation. If so, the Completed tag is set to true and the Executer is invoked. Because the BA-prepare and BA-commit phase have very similar sequences, only one figure is drawn for both phases. Figure 18 is the flowchart of prepare or commit phase.

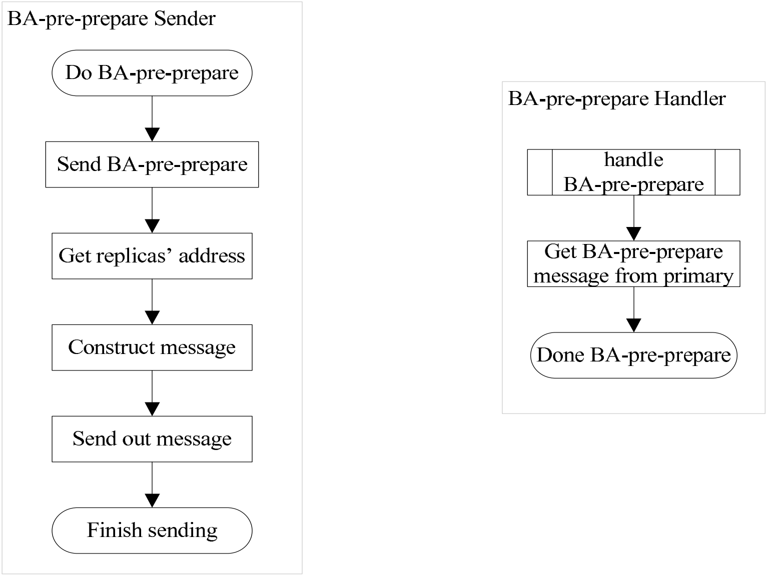

Figure 17: BA-pre-prepare Flowchart Coordinator and Completion Service

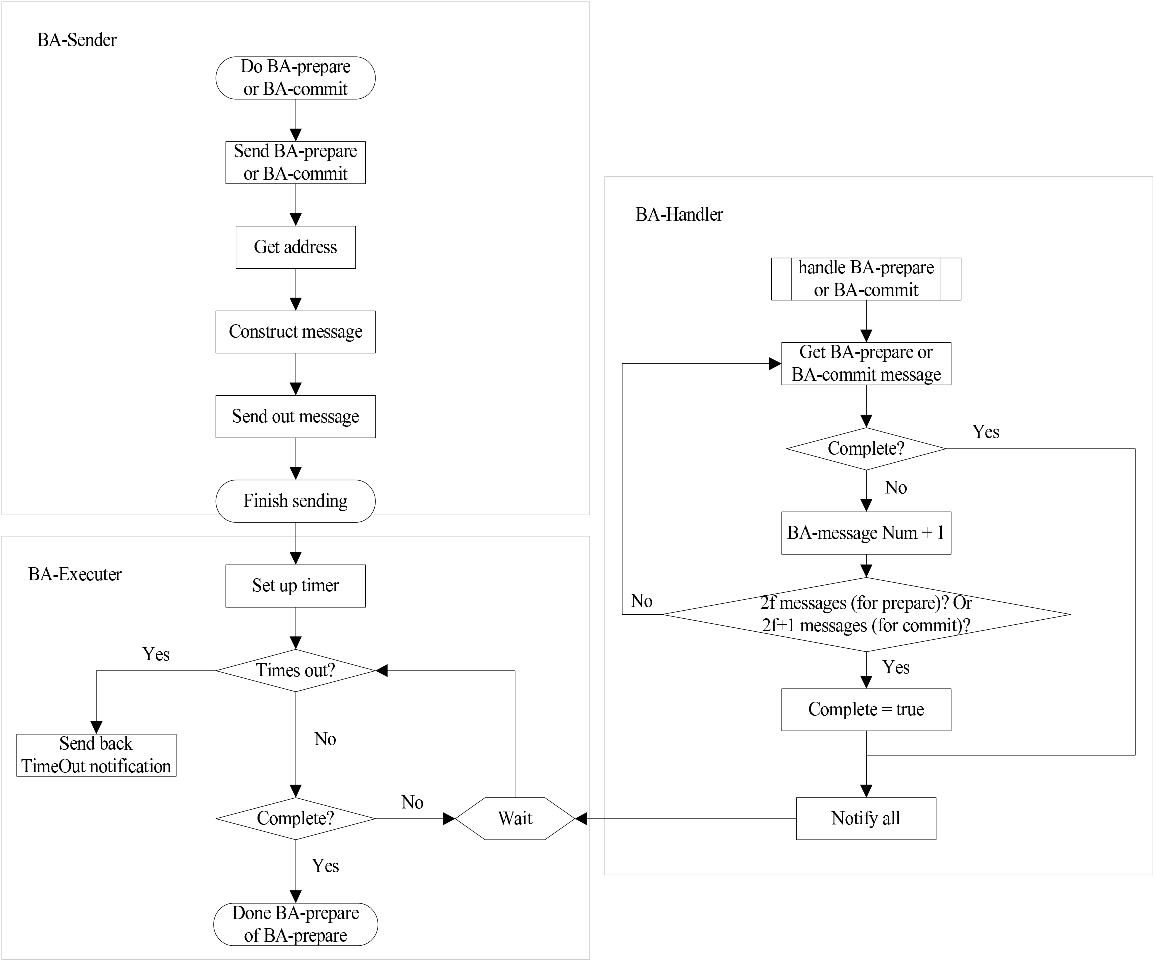
BA-Sender
Do BA-prepare or BA-commit
Send BA-prepare or BA-commit
Get address
Construct message
Send out message
Finish sending
BA-Executer
Set up timer
Times out?
Yes
Send back TimeOut notification
No
Complete?
No
Wait
Yes
Done BA-prepare of BA-prepare
BA-Handler
handle BA-prepare or BA-commit
Get BA-prepare or BA-commit message
Complete?
Yes
No
BA-message Num + 1
No
2f messages (for prepare)? Or 2f+1 messages (for commit)?
Yes
Complete = true
Notify all

Figure 18: Flowchart of BA-prepare or BA-Commit phases

4.1.5 Security Handler

To ensure nonrepudiation, and to prevent impersonation attacks, all messages between all communicating parties including both requests and responses are protected with digital signatures. The messages that could not be verified will be discarded at once without further processing. The Security Handler is responsible to finish all of the work about signing the messages and verifying the signature. This is the second to the last handler before send out a message (the very last one is the Sender).

Before we go into the details about the security handler, I want to clarify what a pivot is. A pivot is the mark point in the message transaction sequence to separate sending and receiving on the client side, or processing a request and producing a response on the server side. For the client, sending procedure is before the pivot. For server, it is inversed.

To perform the digital signature and verification, we need the key pairs. This work has been done before the transaction. We use the keytool utility to generate the key pairs for clients and servers. For more details, refer to Appendix A.

Detail regarding the security handler (MySecHandler) is shown in Figure 19. The functions isClient() and getPassPivot() help us judge whether the message is from the client side or from the server side and whether the progress is beyond the pivot point or not. Based on the result, we execute the signature or the verification corresponding by calling signMessage() or verifySignature() respectively. In MySecHandler, we use the WSS4J engine, a java programmed security engine, to perform digital signing and verification.

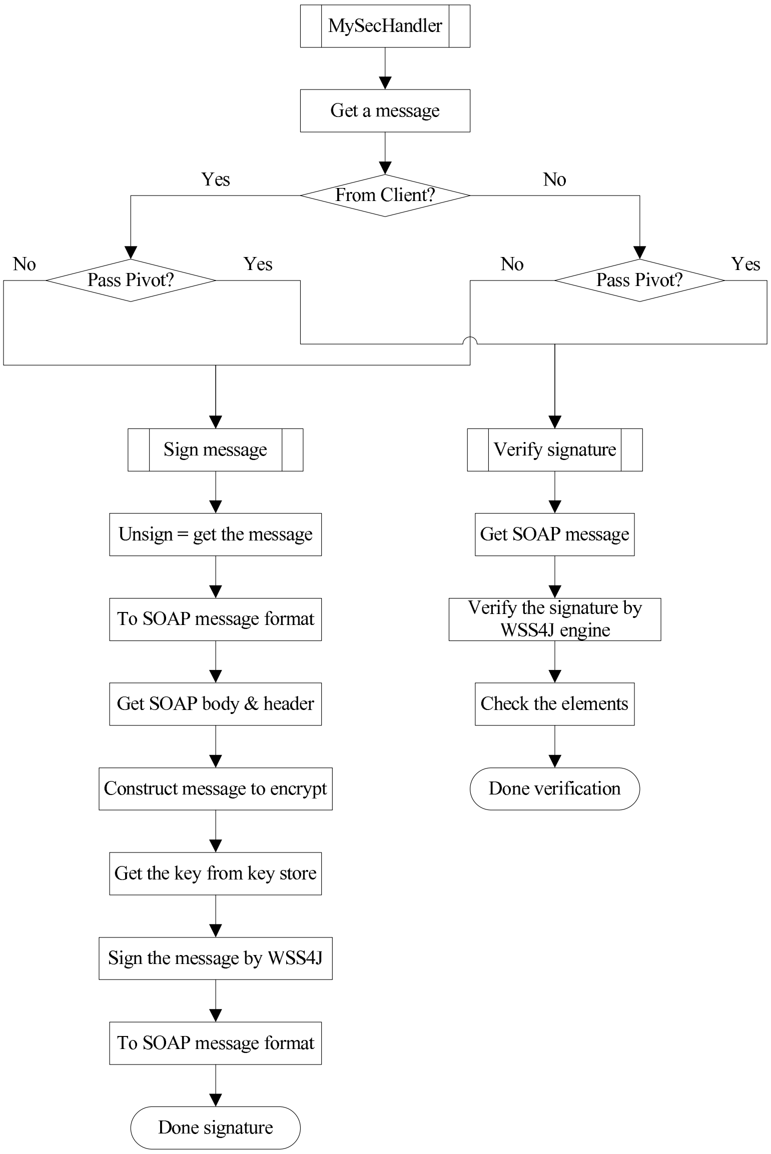

Figure 19: Flowchart of the MySecHandler

4.2 Performance Evaluation

We have implemented the Byzantine Fault Tolerance Distributed Commit in Web Service Atomic Transaction with the exception of the view change mechanisms and integrated it into Kandula project in Java programming language, which implements WS-Coordination and WS-AtomicTransaction frameworks on top of Apache Axis platform.

For performance evaluation, we assess the runtime overhead of our Byzantine Fault Tolerance distributed commit protocol during normal operations. All experiments are carried out on 15 Dell servers which are connected in the 100 Mbps Ethernet. These Dell servers equipped a 2.8GHz processors and 1GB memory running under Suse 10.2 Linux environment.

The test application is the Travel Agent web services application. The client sends the requests to the Travel Agent for booking the flight ticket and hotel among the participants within the scope of a distributed transaction. Travel Agent has four actors as we said before, Client, Initiator, Coordinator and Participants. Because of the equipments limitation, we only enable 4 coordinator replicas to take care a single Byzantine fault. Duo to the stateless, we only used 3 machines to replicate the initiator. For simplicity, the client and the participants are not replicated. All actors including the replicas are running on the different machines. The client invokes a travel agent operation on the Agent web service within a loop without any “think” time between two consecutive calls. 1000 samples are obtained in each run. We increased the number of the participants to get the comprehensive results.

Figure 20 shows the performance of our Byzantine Fault Tolerant Coordination of Web Services Atomic Transaction system implemented in the Travel Agent application, including the distributed commit latency and the end-to-end latency. This is the basic performance for only two participants enrolled in the transaction. The end-to-end latencies for the operation are measured at both of the client side and the initiator side. The latency for the transaction activation and distributed commit are measured at the coordinator replicas sides.

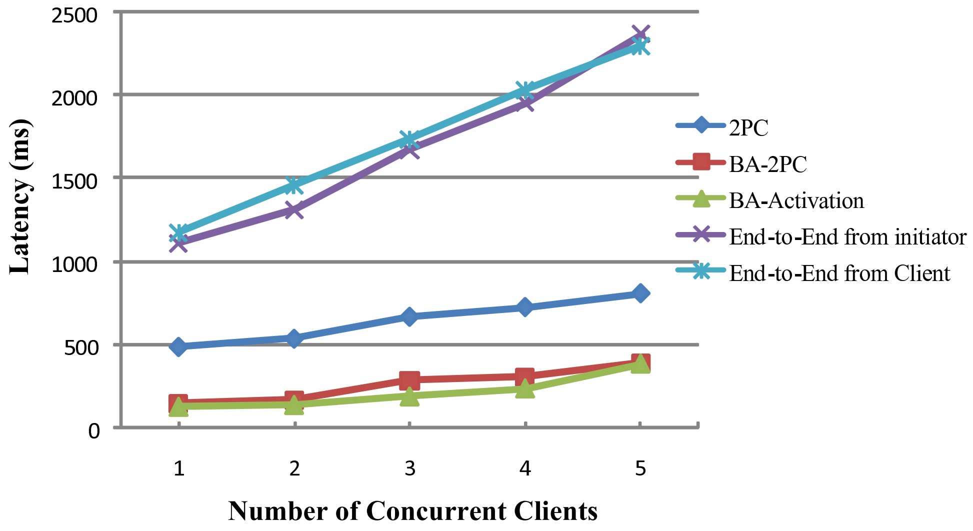

Figure 20: Performance of Our Byzantine Agreement Algorithm

To evaluate the runtime overhead, we compare the performance of our BFT Coordination of WS-AT system with the standard 2PC used in the WS-AT framework with 2 participants enrolled in the transaction. Here all messages exchanged over the network are digitally signed. We compare them from four view points. First one the whole latency for the standard 2PC and BFT enabled 2PC (our system), which is shown in figure 21.

Figure 22 and 23 shows the end-to-end latency comparison from both of the initiator side and the client side.

Obviously, we can see from the previous three figures 21-23 that the latency of the standard 2PC is much smaller than our system because we enable the BFT. For the same reason, the throughput of our system will deduct because of the additional round of Byzantine Agreement algorithm. The last part of the comparison is the throughput. Figure 24 shows the results. From the comparison, we can see that the throughput for transactions in our system is about 30% to 40% lower than those

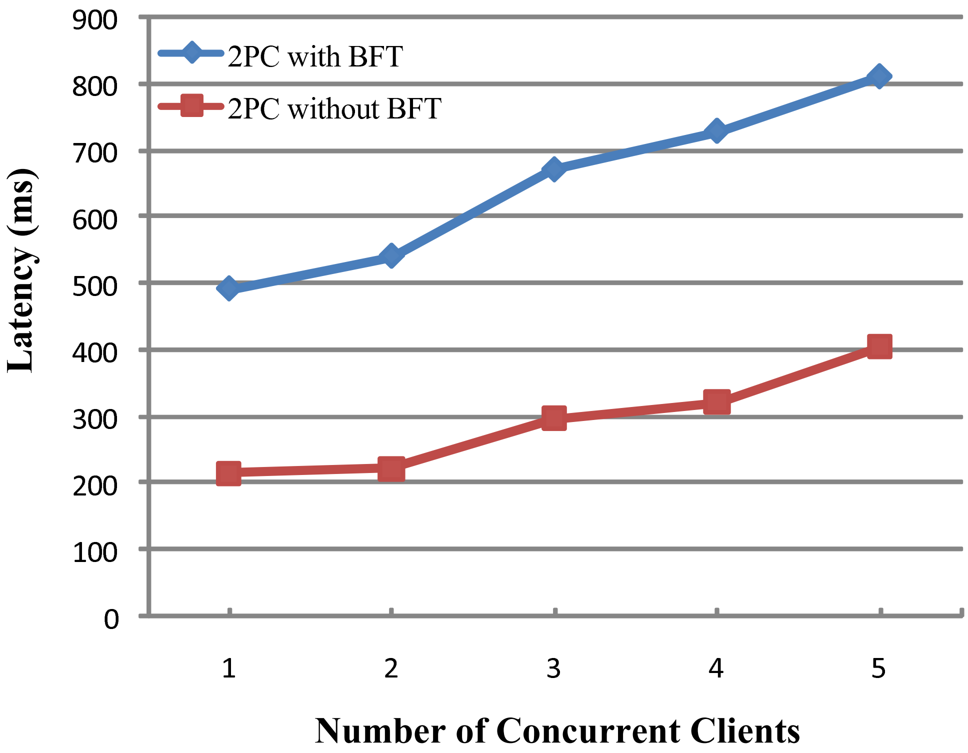

Figure 21: 2PC Latency Comparison

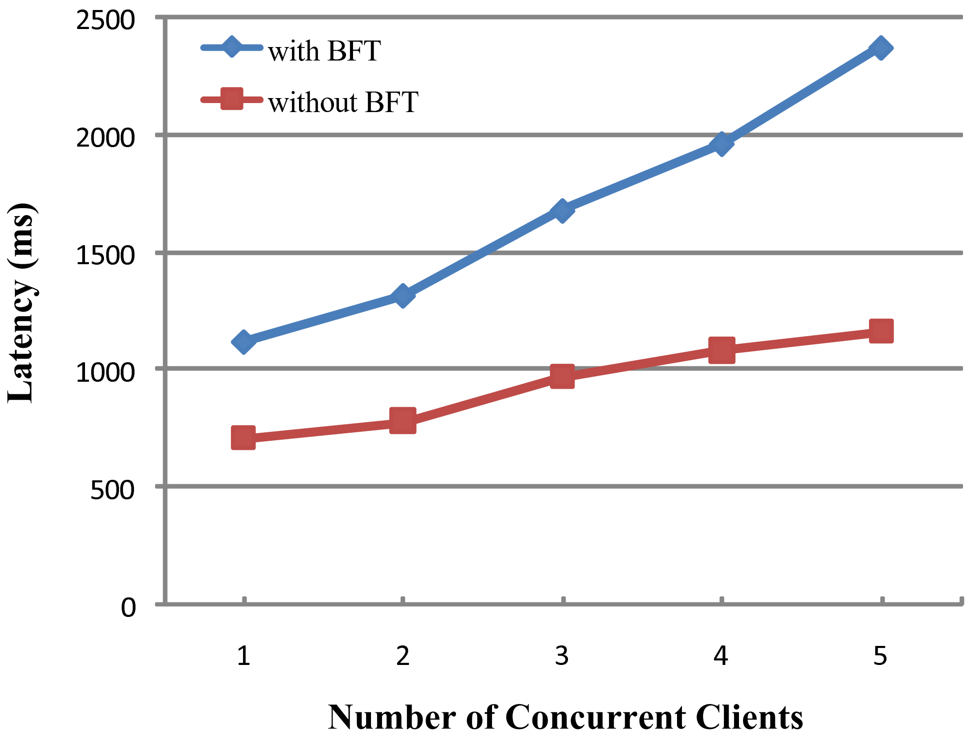

Figure 22: End-to-End Latency from Initiator Side Comparison

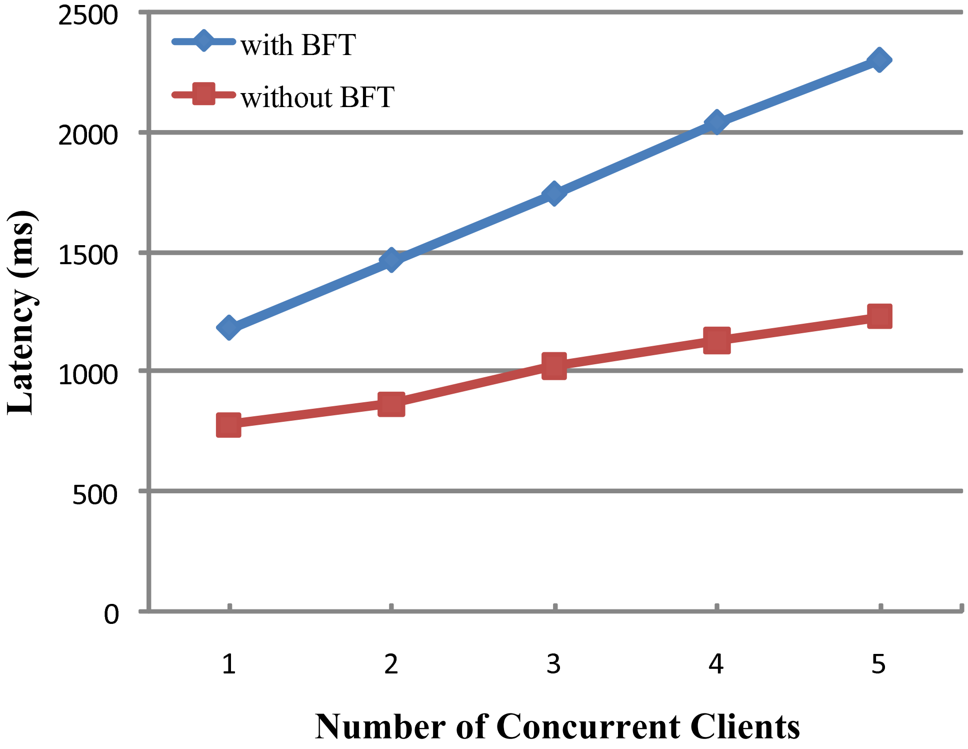

Figure 23: End-to-End Latency from Client Side Comparison

without replication protection, which is quite moderate considering the complexity of the BFT mechanisms.

To better evaluate our system, we did several testing which the number of the participants is increased from 2 to 8. Figure 25 shows the distributed commit latency parallel comparison for different number of participants.

For the whole point of view, figure 26 shows the latency for the 2PC, which is much larger than the latency of the Byzantine Agreement Distributed Commit, because the 2PC is the most costly part in the system.

Figure 27 shows the Byzantine Agreement Activation latency for different number of the participants. Same as previous testing, the number of the clients is from 1 to 5. While the latency for 2PC is lager with the number of participants, the activation latency remains constant because the participants are not involved within the activation.

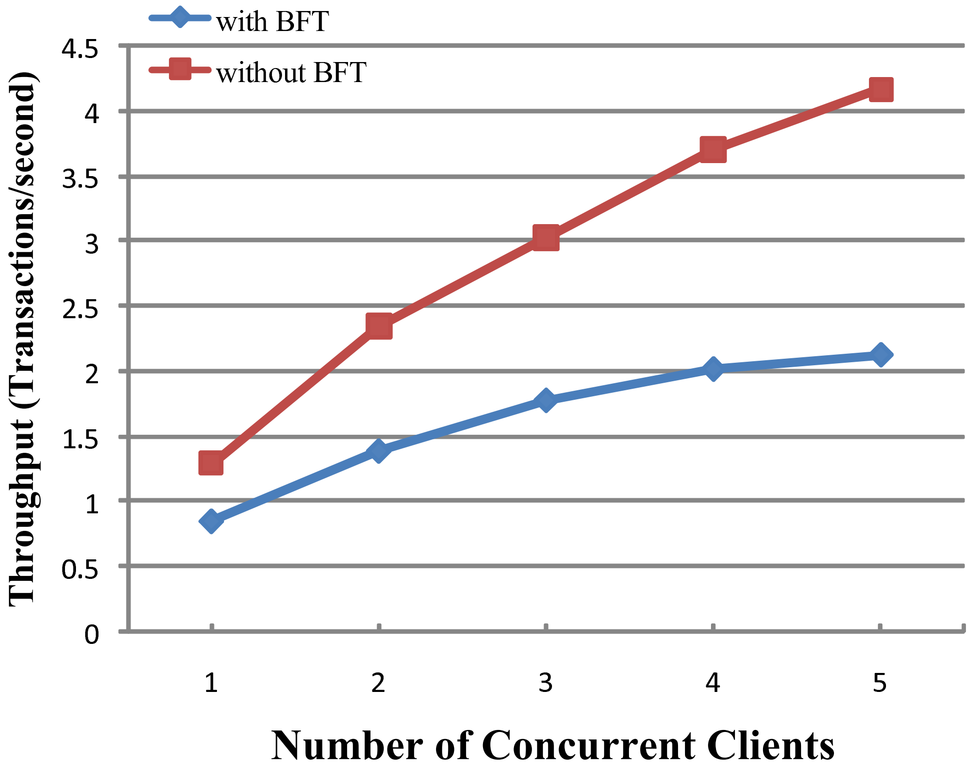

Figure 24: 2 Participants End-to-End Throughput Comparison

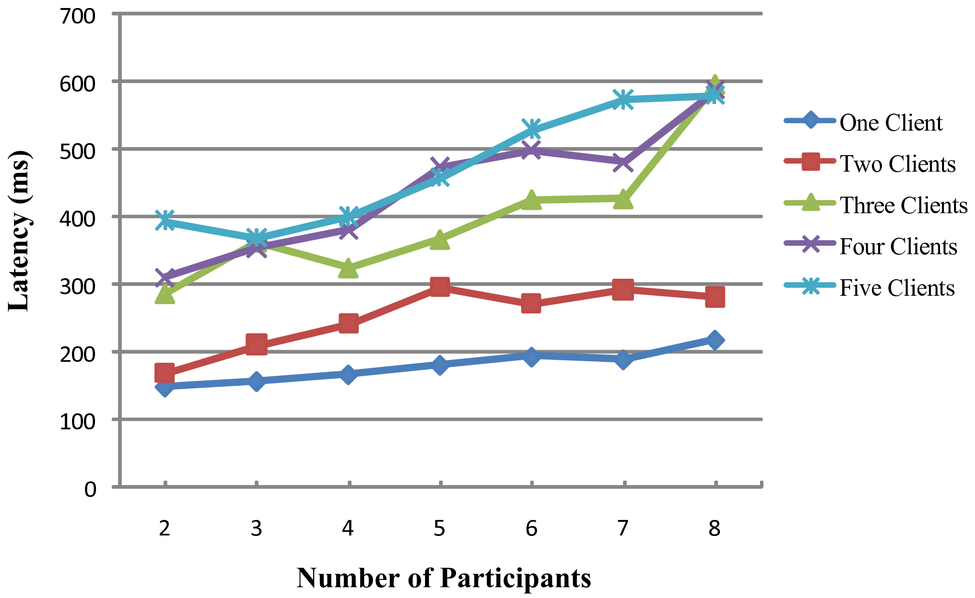

Figure 25: Distributed Commit Latency Comparison

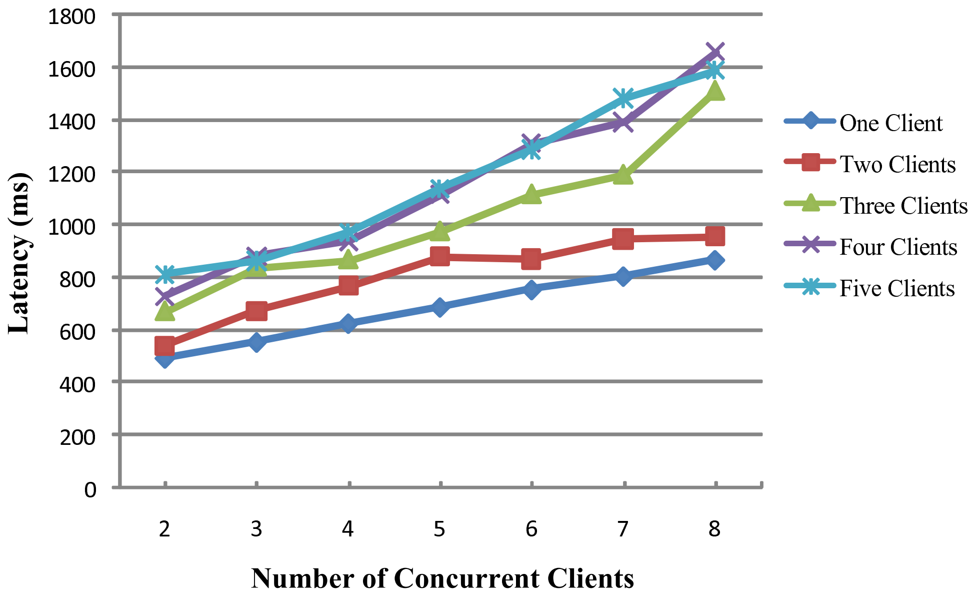

Figure 26: 2PC Latency Comparison

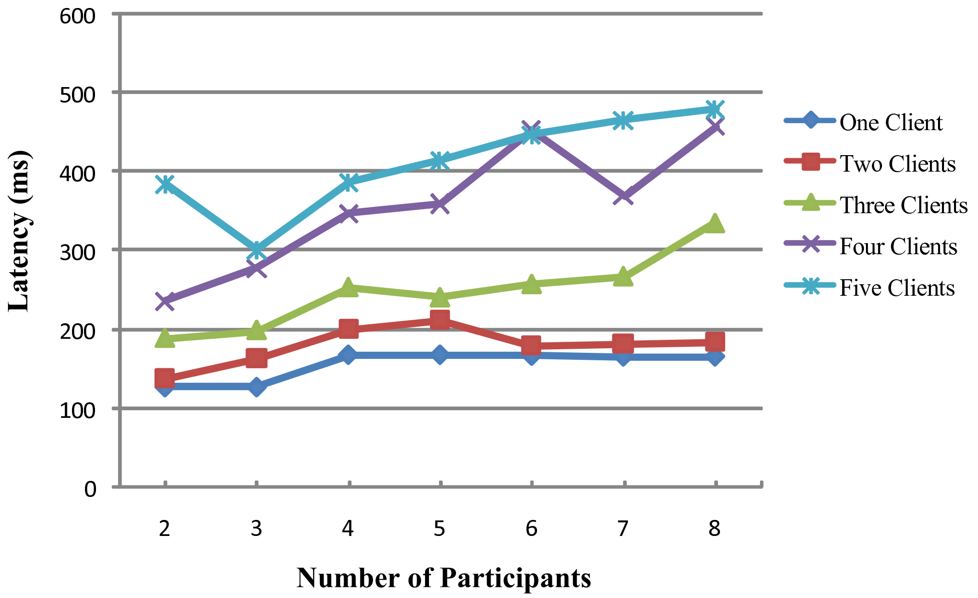

Figure 27: BA-Activation End-to-End Latency Comparison

Figure 28 and 29 show the end-to-end latency from both of the initiator side and the client side. The number of the participants varies from 2 to 8. As we can see in the figures, the end-to-end latency for a transaction is lager by about 400-500 ms for each one more participant. This increase is because of the two Byzantine agreement phases, one for activation and the other for completion and distributed commit. The latency of 2PC is increased by the same reason but there is only one round Byzantine agreement algorithm used in the 2PC, so the increase is smaller. The end-to-end latency both are increased by about 200-400 ms when the number of participants varies from 2 to 8. This increase is mostly attributed to the introduction of the Byzantine agreement phase in our protocol. Percentage-wise, the end-to-end latency, as perceived by an end user, is increased by only 20% to 30%, which is quite moderate.

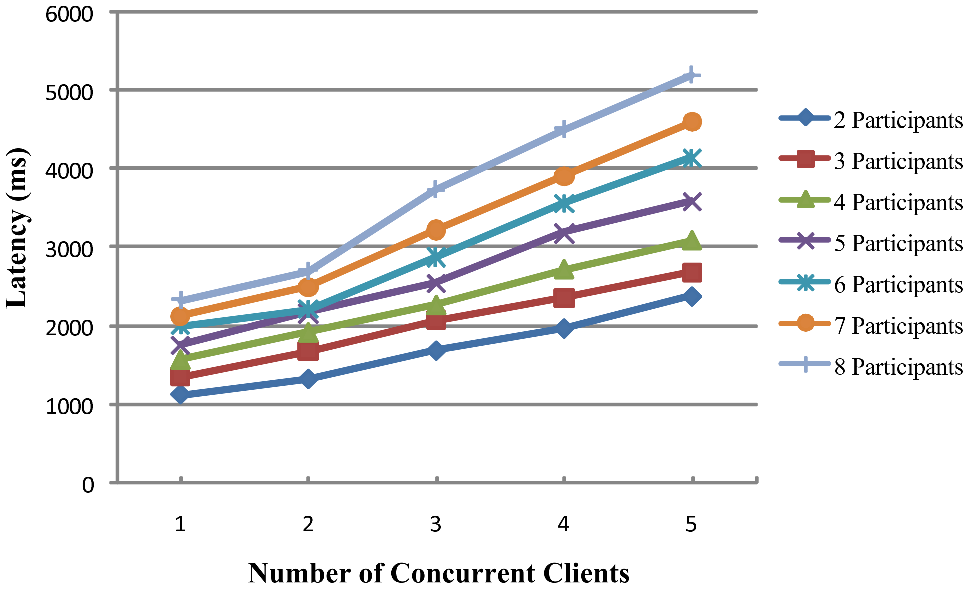

Figure 28: Initiator Side End-to-End Latency Comparison

The throughput of the distributed commit service is measured at the initiator for various numbers of participants and concurrent clients, shown in figure 30.

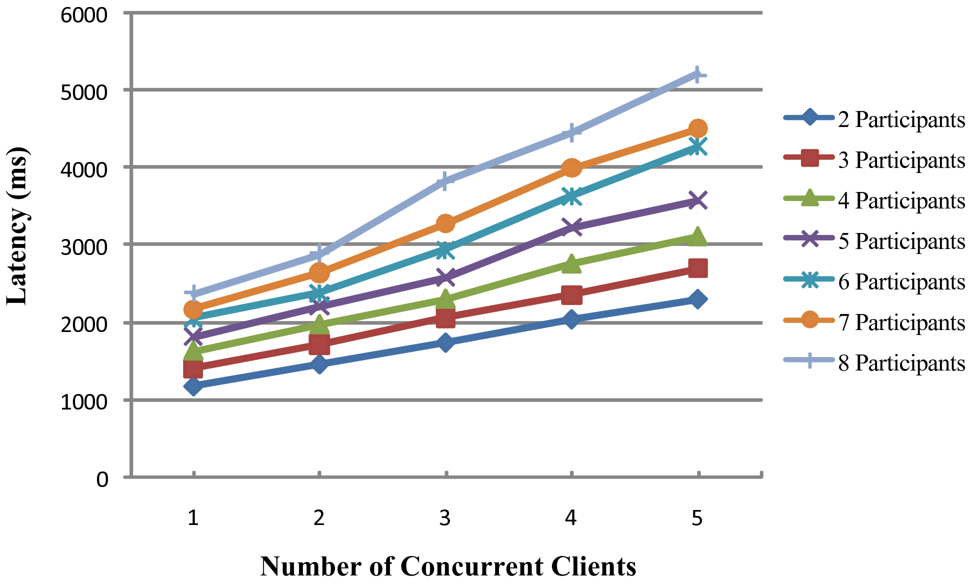

Figure 29: Client Side End-to-End Latency Comparison

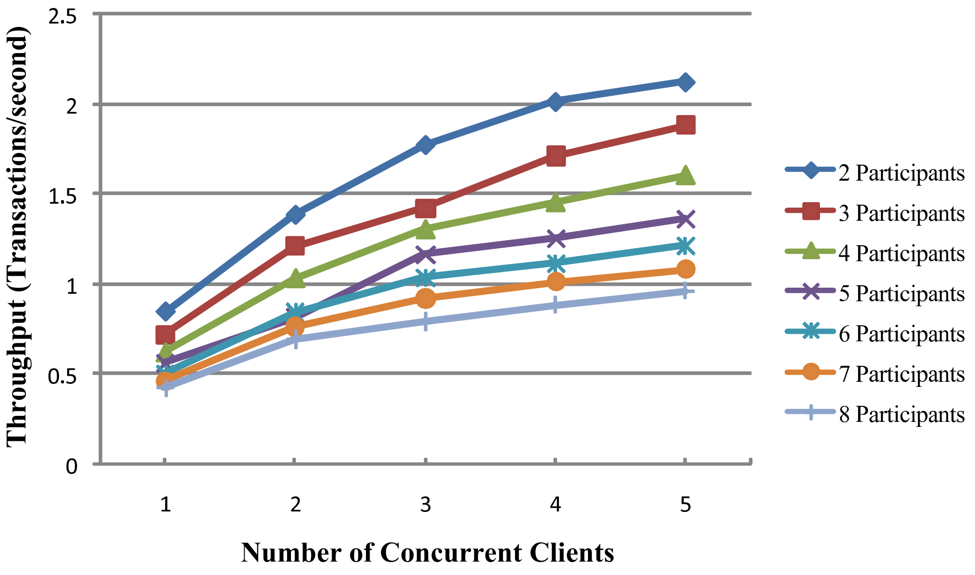

Figure 30: End-to-End Throughput

CHAPTER V

CONCLUSION AND FUTURE RESEARCH

5.1 Conclusions

In this thesis, we presented a Byzantine Fault Tolerant Coordination for Web Services Atomic Transactions system which contains specific mechanisms to ensure atomic transaction commitment. We carefully analysis the types of Byzantine faults that might occur in the distributed transactions and identified the subset of faults that our system can handle. We adapted Castro and Liskov's BFT algorithm [5] to achieve the different object – ensure Byzantine agreement on the outcome of transaction. In our Byzantine fault tolerance system for distributed coordination of Web services atomic transactions, we focus on the protection of the basic services and infrastructures provided by typical TP monitors against Byzantine faults. By exploiting the semantics of the distributed coordination services, we are able to adapt Castro and Liskov's BFT algorithm [5] to ensure Byzantine agreement on the transaction identifiers and the outcome of transactions fairly efficiently. A working prototype is built on top of an open source distributed coordination framework for Web services. The measurement results show only moderate runtime overhead considering the complexity of Byzantine fault tolerance. We believe that our work is an important step towards a highly secure and dependable TP monitor for Web services.

5.2 Future Work

In this thesis, we successfully implement the Byzantine Fault Tolerant Coordination into the Web service atomic transaction to take care of the Byzantine faults. Later, we can also use the similar way to the Web Service Business Activity, which is the other specification based on the WS-Coordination framework for commitment in long running transactions. Or even in the WS-BA-I protocol.

Later, we can do more work concern the business transactions. We identify a research directions regarding Web services coordination for business transactions. WS-BusinessActivity recognized the need for flexible transaction outcomes for business activities, its reliance on compensation transactions might limits its use for some applications. The reservation-based extended transaction protocol [15] seems to be an excellent candidate to augment the compensation-based approach. The basic idea of the reservation-based transaction protocol is that any business activity is carried in two steps. In the first step, a reservation is placed on a set of resources. Depending on the outcome of the reservation step, the coordinator could choose to confirm some reservations while cancel the remaining ones. The use of the extra reservation step eliminates the need for compensation transactions, which could be very expensive and error prone in practice.

BIBLIOGRAPHY

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

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

[3] L. Lamport, R. Shostak, and M. Pease, “The Byzantine generals problem,” *ACM Transactions on Programming Languages and Systems* , Volume 4, No. 3, pp.382-401, July 1982.

[4] C. Mohan, R. Strong, and S. Finkelstein, “Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors,” *Proceedings of the ACM symposium on Principles of Distributed Computing* , pp.89-103, Montreal, Quebec, Canada, 1983.

[5] M. Castro, and B. Liskov, “Practical Byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems* , Volume 20, No. 4, pp.398-461, November 2002.

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

[7] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, et al. (Eds.), *SOAP Version 1.2* , World Wide Web Consortium, 2007.

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

[9] L. Clement, A. Hately, C. Riegen, and T. Rogers, (Eds.), *UDDI Version 3.0.2. OASIS Standard ,* 2004.

[10] L. Cabrera et al., *WS-AtomicTransaction Specification ,* August 2005.

[11] Apache Kandula project, *http://ws.apache.org/kandula/*

[12] S. Frolund, and R. Guerraoui, "e-Transactions: End-to-end reliability for three-tier architectures," *IEEE Transactions on Software Engineering ,* Volume 28, No. 4, pp.378-395, Apirl 2002.

[13] Apache WSS4J project, *http://ws.apache.org/wss4j/*

[14] Apache Axis project, *http://ws.apache.org/axis/*

[15] W. Zhao, L.E. Moser, and P.M. Melliar-Smith, "Unification of transactions and replication in three-tier architectures based on CORBA," *IEEE Transactions on Dependable and Secure Computing ,* Volume. 2, No. 2, pp. 20-33, January-March 2005.

[16] M. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-fault-tolerant middleware for web services applications," *Proceedings of the IEEE Symposium on Reliable Distributed Systems ,* 131-142. 2005.

[17] L. Cabrera et al. *Web Service Coordination (WS-Coordination) Specification ,* August 2005.

[18] K. Rothermel, and S. Pappe, "Open commit protocols tolerating commission failures", *ACM Transactions on Database Systems ,* Volume 18, No. 2, 289-332. June 1993.

[19] A. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, "BAR fault tolerance for cooperative services," accepted by *Proceedings of the twentieth ACM symposium on Operating systems principles table of contents* , Brighton, United Kingdom, pp. 45-58, October 2005.

[20] M. Castro, R. Rodrigues, and B. Liskov, "BASE: Using abstraction to improve fault tolerance," *ACM Transactions on Computer Systems* , Volume. 21, No. 3, pp. 236-269, August 2003.

[21] D. Dolev, and H. Strong, "Distributed commit with bounded waiting," *Proceedings of the IEEE Symposium on Reliability in Distributed Software and Database Systems* , Pittsburgh, pp. 53-60, July 1982.

[22] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Transactions on Database Systems* , Volume. 31, No. 1, pp. 133-160, 2006.

[23] M. Gudgin, and M. Hadley (Editors), *Web services addressing 1.0 - Core* , W3C working draft, February 2005.

[24] B. Hardekopf, K. Kwiat, and S. Upadhyaya, "Secure and fault-Tolerant voting in distributed systems," *Proceedings of the IEEE Aerospace Conference* , Big Sky, Montana, 2001.

[25] C. Pfleeger, and S. Pfleeger, *Security in Computing* , 3^rd ed., Prentice Hall, 2002.

[26] The Open Group, DCE 1.1: Remote Procedure Call, Document Number C706, 1997.

[27] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for Byzantine fault tolerant Services," *Proceedings of*

the ACM Symposium on Operating Systems Principles , Bolton Landing, NY, pp. 253-267, October 2003.

[28] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso, "Middle-R: Consistent database replication at the middleware level," *ACM Transactions on Computer Systems* , Volume. 23, No. 4, pp. 375-423, November 2005.

APPENDIX

APPENDIX A

INSTRUCTION FOR KEY PAIR GENERATION

The keytool utility can generate a key pair. Typically, you must generate two key-pairs to use on both sides of the communication; therefore, execute the keytool with the -genkey option twice, and store each distinct key-pair into a separate keystore.

Step 1: Creating Keystore and Key-Pair

Create two key stores, one for Alice and one for Bob:

```
>keytool -genkey -alias alice -keyalg RSA -keystore alicekeystore -dname ”cn=alice” -keypass foobar -validity 365 -storepass foobar
>keytool -genkey -alias bob -keyalg RSA -keystore bobkeystore -dname ”cn=bob” -keypass foobar -validity 365 -storepass foobar
```

The preceding commands

- Generate separate key-pairs

- Store the key-pairs in separate key stores
- Specify that the RSA algorithm is used as the key algorithm. This is important because otherwise you will see an ”not an rsa key” runtime error from bouncycastle library
- Specify passwords for the keys and the key stores
- Specify the alias/name for each key-pair
- Specify the common name (sometimes referred to as the distinguished name) by which each key-pair will be known within each key store.

To examine the contents of a keystore, we can execute the keytool utility with the -list option. For example, to examine the contents created earlier use:

```
>keytool -list -keystore alicekeystore
```

After providing the password, the output should be like these:

```
Enter keystore password: foobar

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

alice, Nov 25, 2007, keyEntry,
Certificate fingerprint (MD5): 9E:84:0F:0C:D9:B6:B3:6F:31:55:CA:E5:4D:55:C0:BE
```

Now, look at the bob certificate keystore:

```
>keytool -list -keystore bobkeystore
```

```
    Enter keystore password: foobar

    Keystore type: jks Keystore provider: SUN

    Your keystore contains 1 entry

    bob, Nov 25, 2007, keyEntry,
    Certificate fingerprint (MD5): D6:10:24:D5:E4:5F:32:03:19:CA:C6:6B:98:EB:AB:39
```

To examine a key in detail, you can use the keytool utility to display it to the console in RFC 1421 format using the -rfc option, as follows:

```
>keytool -export -keystore alicekeystore -alias alice -storepass foobar -rfc
```

You'll see output on the console similar to the following:

```
    ——BEGIN CERTIFICATE——
    MIIBkTCB+wIER0n6VzANBgkqhkiG9w0BAQQFADAQMQ4wDAYDVQQ
    DEwVhbGljZTAeFw0wNzExMjUyMjQyMzFaFw0wODExMjQyMjQyMzFa
    MBAxDjAMBgNVBAMTBWFsaWNlMIGfMA0GCSqGSIb3DQEBAQUAA
    4GNADCBiQKBgQCfWMml4uE9xrlUAwQsbqSdYvnLa9g51KjSQtOBRE/
    Uy68rg8zmHoYMB7tdgQDP8MQrTukasHDvmzQuSgTvS/iDkw1ZL9g0BRG
    xCDyQuYb4N8VrdAE52pfi57ml/yOspehrPbYKNJuyoKVrM/6SDmMGwL9
    /7fzTB7i9zzZWTfqvqwIDAQABMA0GCSqGSIb3DQEBBAUAA4GBACq6Q
    A0VJc97yutdVB2G08HuBfycOogHRQCUW6KBGSIe3YtC9ZQH+C3xfaUV
    Rr97MkEUtInmNZidmGias7RybKPb9Q7ZDKb4XDzvekl8MWzRaW8eg6gx3
    woyFdJHKYSLBFTHsMnffM/VU6jBCjeu/GVdeQDctqyZ4+83x/hK7LJx
```

```
——END CERTIFICATE——
```

Do the same operation for bob's key store, as follows:

>keytool -export -keystore bobkeystore -alias bob -storepass foobar -rfc

The output:

```
——BEGIN CERTIFICATE——
MIIBjTCB9wIER0n6aDANBgkqhkiG9w0BAQQFADAOMQwwCgYDVQQ
DEwNib2IwHhcNMDcxMTI1MjI0MjQ4WhcNMDgxMTI0MjI0MjQ4WjAO
MQwwCgYDVQQDEwNib2IwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMI
GJAoGBAITUmI5/EJZ4NUNKrtoDZ3lVMys0sNzLFiI5f1BrdHFSdv1rsmn
dFZ+9OZi0xDMOGFtPzCreZdoGUym3y20hX7+UzucLcgecdZhm4zcByNE
GwR2KQ0xzyu9oreiABrIvC0cmU35UDyFPy61stE7YO7GEqc+yspDWh5w
Pejc1Q1KDAgMBAAEwDQYJKoZIhvcNAQEEBQADgYEAGUN/IHUwiN
ULEfTEalvltLhFbyCvagk/e0YksRY7q33kpL49w1WdGOtJgwe+ITE9+Rb2
bBe2InUix+f7EuCj4REoGl55PHQ0OdiKxw8HAuXR9hFDO6o/tgEqD/Mq
rmnNs56m3N67LXTNqtnRIq/i8hRLqwJkL3V0y9uOIT8AstA=
——END CERTIFICATE——
```

Step 2: Self-Signing Certificates

Keys are unusable unless they are signed, but you can use the keytool to self-sign them (for testing purposes only), as follows:

Self certify the keys for Alice and Bob:

>keytool -selfcert -alias alice -validity 365 -keystore alicekeystore -keypass foobar -storepass foobar

>keytool -selfcert -alias bob -validity 365 -keystore bobkeystore -keypass foobar -storepass foobar

Step 3: Exporting Certificates with the Keytool Utility

After generating and self-signing the keys/certificates and storing them in the keystores, import each public key into the other key's keystore. This requires two steps: exporting the public key to a certificate file and importing the certificate to the other keystore.

To export the public key to a certificate file, use:

```
>keytool -export -alias alice -keystore alicekeystore -keypass foobar -storepass foobar -file alicecert
>keytool -export -alias bob -keystore bobkeystore -keypass foobar -storepass foobar -file bobcert
```

You should see the responses:

```
    Certificate stored in file <alicecert>
```

and

```
    Certificate stored in file <bobcert>
```

You can also use the keytool utility to display the contents of the certificate file using the -printcert option, as follows:

```
>keytool -printcert -file alicecert
```

The output will look like:

```
Owner: CN=alice
Issuer: CN=alice
Serial number: 4749fa57
Valid from: Sun Nov 25 17:42:31 EST 2007 until: Mon Nov 24 17:42:31 EST 2008
Certificate fingerprints:
    MD5: 9E:84:0F:0C:D9:B6:B3:6F:31:55:CA:E5:4D:55:C0:BE
    SHA1: 0E:95:8A:7D:F3:53:99:29:74:A7:32:7B:CC:06:37:46:2D:3F:86:1F
```

The exported certificate contains the public key and distinguished name given to the certificate (in this case, alice).

Step 4: Importing Certificates into Keystores

To import a public certificate into the keystore of the private key, issue the command:

Import key certificate for Alice into Bob's keystore, and vice versa:

```
>keytool -import -keystore alicekeystore -alias bob -keypass foobar -storepass foobar -file bobcert
>keytool -import -keystore bobkeystore -alias alice -keypass foobar -storepass foobar -file alicecert
```

The output looks like:

```
Owner: CN=alice
Issuer: CN=alice
Serial number: 4749fa57
Valid from: Sun Nov 25 17:42:31 EST 2007 until: Mon Nov 24 17:42:31 EST 2008
Certificate fingerprints:
        MD5: 9E:84:0F:0C:D9:B6:B3:6F:31:55:CA:E5:4D:55:C0:BE
        SHA1: 0E:95:8A:7D:F3:53:99:29:74:A7:32:7B:CC:06:37:46:2D:3F:86:1F
```

Answer the following question:

```
        Trust this certificate? [no]: yes
        Certificate was added to keystore
```

Now that the certificate has been imported into the alice key's keystore, you can reexamine the contents of the keystore using the keytool utility with the -list option, for example, to list information of alicekeystore as follows:

```
>keytool -list -keystore alicekeystore
```

```
Enter keystore password: foobar
```

After entering your password you'll see the following output:

```
Keystore type: jks
Keystore provider: SUN

Your keystore contains 2 entries
alice, Nov 25, 2007, keyEntry,
Certificate fingerprint (MD5): 9E:84:0F:0C:D9:B6:B3:6F:31:55:CA:E5:4D:55:C0:BE
bob, Nov 25, 2007, trustedCertEntry,
Certificate fingerprint (MD5): D6:10:24:D5:E4:5F:32:03:19:CA:C6:6B:98:EB:AB:39
```

As the preceding examples illustrated, there are now two entries in the alice-key's keystore. The first, with the alias alice, is identified as a key entry. The second entry, with the alias bob, is imported from the certificate file.

At this point you have performed sufficient key management tasks to use the key pairs to perform WS-Security tasks using the Apache Web Services Security for Java framework.

Note that one of the keystore is used by the sending side (client) to retrieve the receiver's public key certificate. The messages sent to the receiver will be encrypted using the receiver's public key. The other keystore is used at the receiving side to retrieve its own private key to decrypt messages encrypted using its public key.

APPENDIX B

USER GUIDE

Instruction:

1. Unpackage the secure2pc and tomcat-secure2pc.
2. Go to the secure2pc directory and do maven by using this command
   >maven
3. Go the *secure2pc/src/samples/agent/* directory and build the example by
   >ant dist
4. Copy the agent.jar from *secure2pc/src/samples/agent/build/* to
   tomcat-secure2pc/webapps/axis/WEB-INF/lib/
5. Copy kandula-0.2-SNAPSHOT.jar from *secure2pc/target/* to
   tomcat-secure2pc/shared/lib/
6. Generate the key pair. Please go to Appendix A for detail.
7. Copy the key store to *tomcat-secure2pc/webapps/axis/WEB-INF/classes/* directory and *secure-2pc/ws/lib*

8. Start tomcat on every nodes by going to tomcat-secure2pc directory and running *>./startup.sh*

9. Start the tcp monitor on every nodes.

10. Go to *secure2pc/ws/bin/* directory and run *>./run.sh test.Tester*