2010

A Light Weight Fault Tolerance Framework for Web Services

Srikanth Dropati
Cleveland State University

A LIGHT WEIGHT FAULT TOLERANCE FRAMEWORK FOR WEB SERVICES

SRIKANTH DROPATI

BACHELOR OF TECHNOLOGY (B.Tech)

COMPUTER SCIENCE AND ENGINEERING

Jawaharlal Nehru Technological University, India

May, 2005

Submitted in partial fulfillment of the requirement for the degree

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

July, 2010

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. Nigamanth Sridhar

_____

Department/Date

To

My Family and Friends …

ACKNOWLEDGEMENTS

First, I would like to express my gratitude and thanks to my thesis advisor, Dr.Wenbing Zhao for his enthusiasm, patience and valuable guidelines in my research work. I feel fortunate to work under the guide lines of such an expert, whose guidance not only helped me in completion of my research but also helped me built confidence to handle the real-time career challenges.

I would also like to thank my committee members and my department advisors Dr. Yongjian Fu and Dr. Nigamanth Sridhar for their valuable suggestions and support through my studies at the university.

I would like to thank Dr.Yongjian Fu for being my advisor throughout my study program at the university and for supporting me financially through the KCIDE project.

I would also like to thank my software engineering lab mates who have helped me with my research.

Finally, I would like to thank all my family and friends for their continued support and encouragement.

A LIGHT WEIGHT FAULT TOLERANCE FRAMEWORK FOR WEB SERVICES

SRIKANTH DROPATI

ABSTRACT

The increased usage of web services by many of the corporate industries to exchange their critical information over World Wide Web has directly impacted the need for high availability of web services. So in this work of ours we designed and developed a light weight fault tolerance framework for web services, by strictly biding ourselves to the design specifications of web services. We developed our framework by extending the open source implementation of Web services reliable messaging specifications. Our framework provides fault tolerance capability using the replication strategy, and can easily be reverted back to basic point to point reliable message specifications implementation dynamically upon availability of resources. We used a customized consensus solving algorithm to achieve and maintain consistency among the replicated systems. The message patterns that are used to exchange the data are very much bided to the message specifications of web services. Our framework does not use any proprietary protocols for transmission of messages over the network. We also carefully tuned our framework for enhanced performance by techniques like batching, and proved from our performance results that our framework is optimal and has very less run time overhead.

TABLE OF CONTENTS

LIST OF FIGURES

ACRONYM

I. LFT – Light Weight Fault Tolerance Framework

II. BFT – Byzantine Fault Tolerance

III. WSRM – Web Services Reliable Messaging Specifications

IV. XML – Extensible Markup Language

V. WSDL - Web Service Definition Language

VI. SOAP – Simple Object Access Protocol

VII. UDDI – Universal Description, Discovery & Integration

VIII. RMD – Reliable Messaging Destination

IX. RMS – Reliable Messaging Source

X. LAN – Local Area Network

CHAPTER I

INTRODUCTION

Our primary goal with this work is to design and develop a light weight fault tolerance framework for web services by strictly abiding to the design specifications of web services. Our framework revolves mostly around some of the renowned technologies like web services, fault tolerance and its common techniques and WSRM specifications. So before jumping in to the design specifications of our framework, here is a brief introduction to the technologies that have been used to develop our framework.

1.1 Web Services

Web services, the most commonly used technology in the present day computer world for exchange of critical messages, is formally defined as a group of relevant operations on a process, which could be accessible over a network. To be more precise, they are web based applications that use XML-based documents formatted according to SOAP rules, standards and transport protocols to exchange data with clients and with other web services [26]. Web services are often defined in a machine process able format termed as Web services description language (WSDL) [26].Web Services are known for their ability to operate both on intranet and Internet environments. With the advent of web services the scope of IT has considerably. Web services capability to communicate with services on heterogeneous platforms overcomes the platform dependency hurdle.

1.1.1 Architecture of Web Services

The high level architecture view of web services is shown in Figure 1. Web services from their definition [26] are defined as an interoperable machine to machine interaction over a network. The basic definition of web services seems to be simple but down the lane they offer many optimal solutions for the exchange of the critical messages over the network especially over World Wide Web, by overcoming the overhead incurred due to platform dependency and network congestion.

Web services have reduced the overhead and cost incurred by the previously existing remote procedure calls (RPC) technologies like CORBA, DCOM by providing support to exchange of information asynchronously. Compared to the previously existing remote procedure call technologies that are well-known for their high resource consumption, heavy weight architecture, and less robustness, the web services could successfully provide an optimal solution with light weight architecture and comparatively less consumption of resources due to its asynchronous nature of message exchange.

Web services architecture mainly comprises of 3 major terms namely UDDI, WSDL and SOAP protocol, and a network medium like LAN (Local Area Network) or World Wide Web (internet) to transmit messages.

ACRONYM's used in Figure 1:

WSDL - Web Service Definition Language

SOAP – Simple Object Access Protocol

UDDI – Universal Description, Discovery & Integration

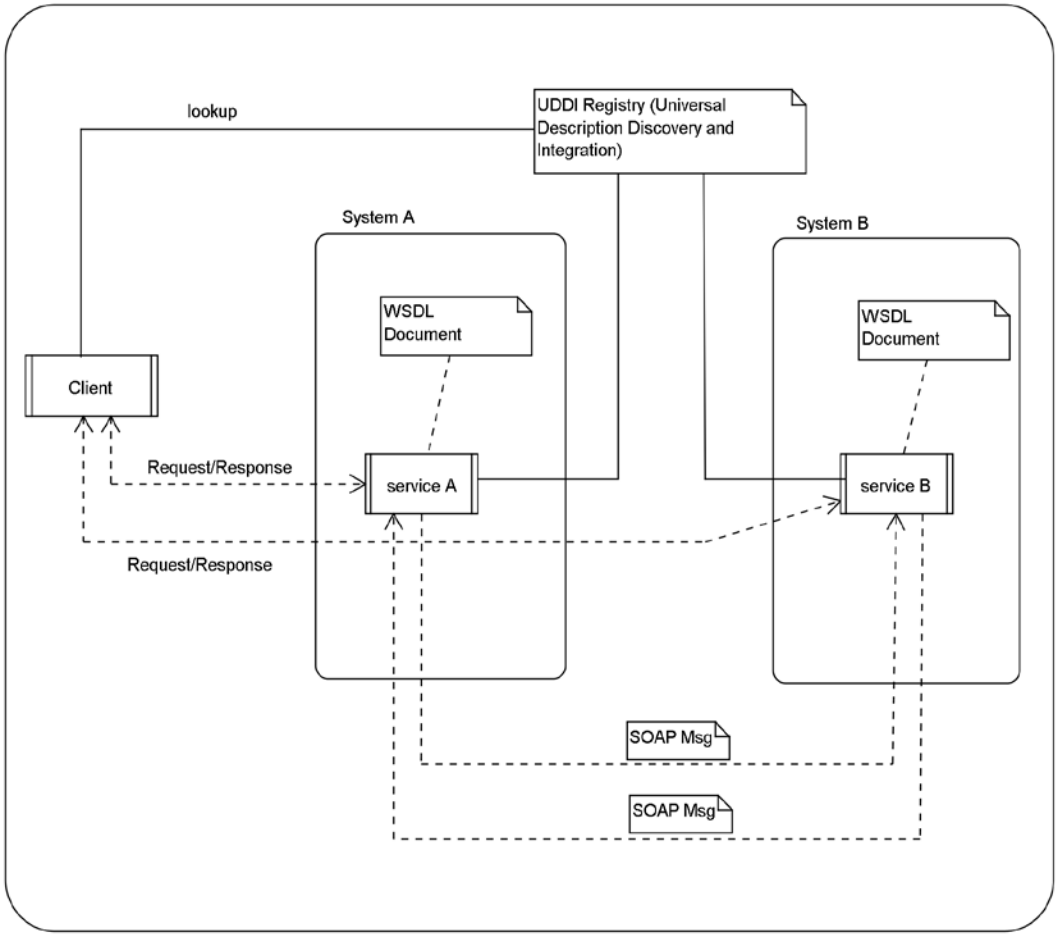


Figure 1. High level Architecture of Web Services.

1.1.1.1 Universal Description, Discovery and Integration (UDDI)

This UDDI registry is used as a common place to record the information belonging to a web service like the owner of the service, the functions offered and supported by the web service including the documentation that is necessary to invoke the service. To better classify the information stored at UDDI registry, it uses 3 major components namely white pages, yellow pages and green pages. To dig in deep about these components, white pages store the address and contacts, yellow pages are used to store the information

regarding industries like its standards etc and finally green pages are used to store the technical information regarding the actions, functions handled and provided by the listed services.

1.1.1.2 Web Service Definition Language (WSDL)

WSDL is considered as a structured way to define the web services in XML language usually containing the information necessary by a client/source to invoke the actions on the service, binding information about the transport protocol to be used and the address and port information for locating the specific web service over the network. WSDL document is defined using XML language mostly containing the information about the publicly available functions from this service and a definite ways to reach and invoke the functions of the service. It also possesses the detailed information about the arguments, including their data types to avoid all the confusions on the client side. WSDL document also highlights the expected results type so that appropriate result handling mechanism could be provided at client side.

1.1.1.3 Simple Object Access Protocol (SOAP)

SOAP is a lightweight protocol, written using XML language, often used to exchange information between heterogeneous platforms. The message packet that is structured according to SOAP is termed as SOAP envelope and it comprises of header and body blocks. Of the two components header is optional block and body is considered to be mandatory. SOAP is compatible with almost all the available network protocols like SMTP, FTP and HTTP, of which SOAP over HTTP is considered to be more efficient

and a prescribed one according to Web services specifications.

Considering the inbuilt support to exchange of data asynchronously and thereby supporting loosely coupled concept, many leading corporate companies have moved on to use Web services as a basic means to exchange their critical information over World Wide Web. Almost all of the languages have come up with the supportive API's (Application programming interface) for web services, to provide a means for the request broker implemented on a particular language to invoke the respective web service by just looking it up through the UDDI registry and by decoding the WSDL document, without a need for man to man interaction to establish environment.

1.2 WSRM Specifications

Web Services Reliable Messaging(WSRM) specifications are the specifications put forward by major organizations like IBM, BEA, Microsoft and TIBCO, to allow messages to be delivered reliably between distributed applications in the presence of software component, system or network failures [14].

1.2.1 Need for the Web Services Reliable Mechanism Specifications

As the web services often operate and communicate over the network they use the common network protocols like HTTP (Hyper Text Transfer protocol) or FTP (File Transfer Protocol), to transmit the messages between end-points. Though the transfer protocols engaged in this process guarantee the successful transmission of messages between the network end points, but they lack the application level message transmission

consistency, which thereby affects the correctness of the message delivery information report and reliability.

Consider two business processes A and B that relies on web services for communicating and for exchanging the critical information. As web services technology is used to exchange the data, SOAP standards are used to form the message packets. So naturally the transportation of the SOAP messages is carried out using either a HTTP protocol or a FTP protocol. The business process which would be implemented using one of the available web services frameworks of a given language, would generally wrap the low level transport layer with a framework helper classes layer. Though the existing network protocols promise a source to destination end point to endpoint message transfer, they would not provide application level end point message transmission reliability.

Consider a situation of message exchange triggered from process A, transferred to process B and a message status report to be generated at process A for further usage. So, the process A generates the SOAP message using the API and libraries (related to web services provided by the programming language on which process A is implemented), and would transport the message to the process B using the network channel like World Wide Web.

Normal Operation

Once the Process A triggers the transmission of message packet to process B, the packet goes through the transport channel of the framework in which process A is im-

plemented and thereby the network protocol would handle its further transmission i.e., an endpoint to endpoint transmission is carried out by network protocol and from the destination network endpoint the message is transported to process B. In the same time once the destination end point receives the message packet an acknowledgement is generated and sent back to process A through source network endpoint.

Inconsistent Case

Consider the situation where destination network end point has received the application message and before accepting the message assume Process B has crashed. Then, though the message is lost on network after destination end point of network and before the process B could receive it, a success status message about the application message is acknowledged back to source end point of network by the transport protocol. A transport protocol all it cares is a transport channel end point to end point transmission but not the application level message transmission. So it would be dangerous for the process A to assume that the message sent will reliably reach the process B.

To avoid above situations and to provide reliability among such kind of message transactions, the major organizations namely BEA Systems, Microsoft Corporation Inc, IBM and TIBCO Software Inc have proposed WSRM [14] specifications to guarantee an application level message transmission. These specifications with very less effort support reliable nature for web services i.e., it helps the web services to exchange their messages with clients or other services reliably. A concept of sequence, defined in detail in WSRM specifications, is used to provide reliable nature among web services.

To provide reliable nature for existing web services, for allowing them to converse reliably, there is no need to alter them in any way instead, WSRM specifications could be added as a whole new plug-in to the pre-existing web service. The WSRM specifications guarantees application to application reliability through re-transmission of the lost messages and at the same time it also takes enough care to avoid duplicate transmission of messages. An in-depth functionality of WSRM specifications is discussed in the next section.

1.2.2 WSRM Specifications In-Detail

The protocol defined by WSRM specifications does not depend on any proprietary transport protocol. It can be implemented using any available transport network protocol. The protocol adopts many of the web services design specifications like, to form and exchange message packets, identifying and obtaining the service endpoint addresses and policies.

WSRM specifications classify the end points between which the reliable message exchanging takes place as reliable message source(RMS) and reliable message destination(RMD). WSRM also proposes the new term called “sequence” to reliably exchange the messages between the RMD and RMS. Sequence is a unique id proposed through a series of sequence creation messages exchanged between RMS and RMD. It is included in every application message to reliably exchange messages. An in-detail explanation about the functionalities carried out at RMD and RMS with sequence concept is defined in WSRM specifications, so to avoid repetition its elaboration is skipped here.

WSRM specifications also facilitate some of the following message delivery assurances, which could be combined further for stronger performance.

AtmostOnce: This delivery assures that message would be delivered at most once, avoiding the duplication of the messages at the receiver side. If failed produces an error at atleast one end point. Though the resource usage hold is less and performance is high compared to other mechanisms, the possibility to miss delivery of messages is also high in a given sequence of messages.

AtLeastOnce: This delivery assurance would make sure to deliver message at least once to the destination or else an error would be produced at the end point. Some messages may be delivered more than once.

ExactlyOnce: Every message would be delivered exactly once at the end point without any duplication or else an appropriate error would be produced. This delivery assurance is the logical “and” of the two prior delivery assurances [14].

InOrder: This assures that the messages would be delivered in the order they were sent. This delivery assurance could be combined with other delivery assurance to provide stronger delivery assurance. The combination of the Exactly Once and In Order assurances is considered to be the strongest assurance.

The Apache software foundation has provided an open source implementation of the WSRM specifications using java programming language, termed as Apache Sandesha2[1] module. This is provided as a plug-in module which could be added to any given web service framework of java for providing the web services to offer reliable services to its clients. The high level component diagram of Apache Sandesha2 is shown

in Figure 2, with Sandesha2 module being engaged on both client and server side to offer reliable message exchanging between the client and the server.

Note, in the Figure 2, a single channel of send and deliver is presented. But in reality a new reliable channel establishment is purely based on the number of sequences that are established between the given client and the server. Since an in-detail sequence management and channel establishment is explained in WSRM specifications [14], we skipped it here to avoid repetition.

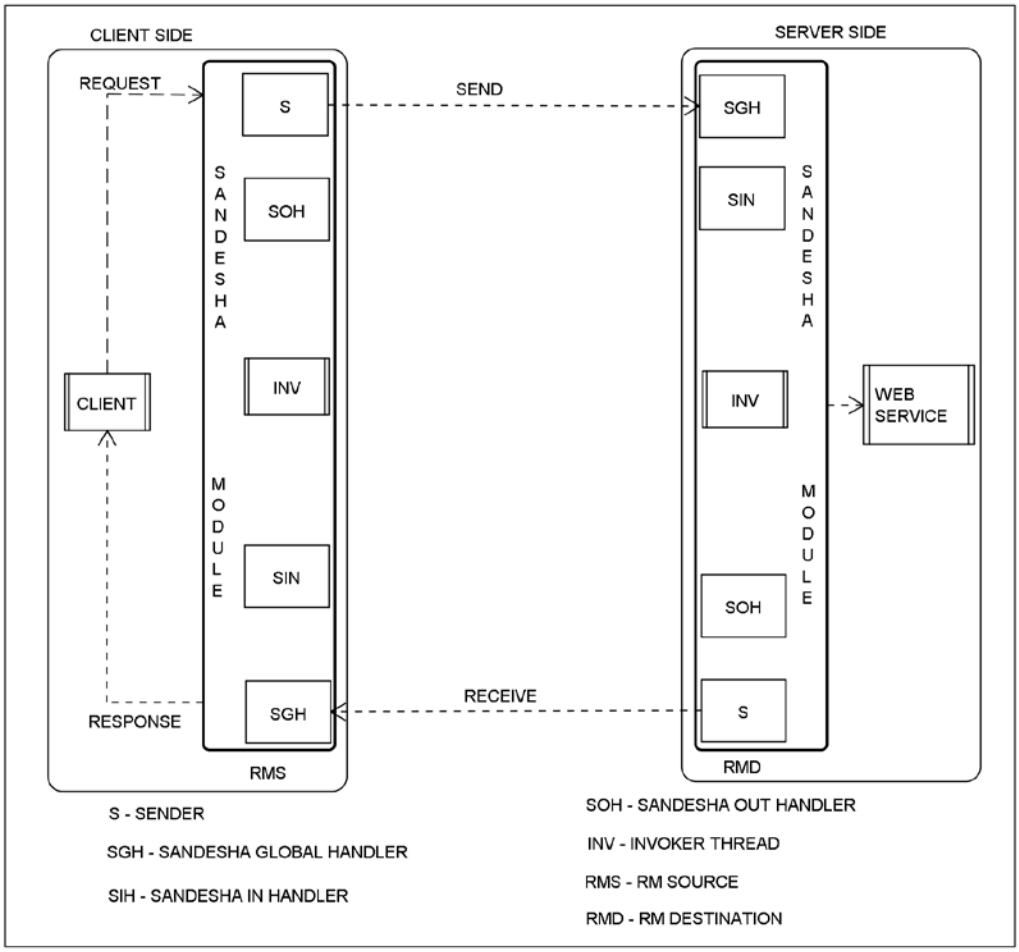


Figure 2. High level component diagram of Apache Sandesha2.

The Sequence diagram pattern for Apache Sandesha2 is shown in Figure 3. To further

understand the message flow between the Sandesha2 modules, engaged on client and server side to support reliable message exchange between them, a sequence diagram is shown in Figure 3. The Figure 3, indicates how a client side and server side engaged Sandesha2 modules would initiate a sequence oriented reliable channel and acknowledge themselves about the missing of a particular application message within a given sequence and trigger retransmission of the message. Once they both confirm the proper transmission of all messages from client to server they would terminate the appropriate sequence.

WSRM specifications use acknowledgements to guarantee a reliable exchange of messages between two processes. An acknowledgement corresponding to each successfully delivered message is maintained at the sender side for further usage like to provide a final status report. If an acknowledgement is not received for a message sent in a given period of time, it assumes the loss of message and retransmits the message. The retransmission of the message is carried on till an acknowledgement corresponding to the sent message is received. Figure 3, clearly indicates how the sender would guarantee reliability by resending the message to the destination point, if the corresponding acknowledgement is not received in a given period of time.

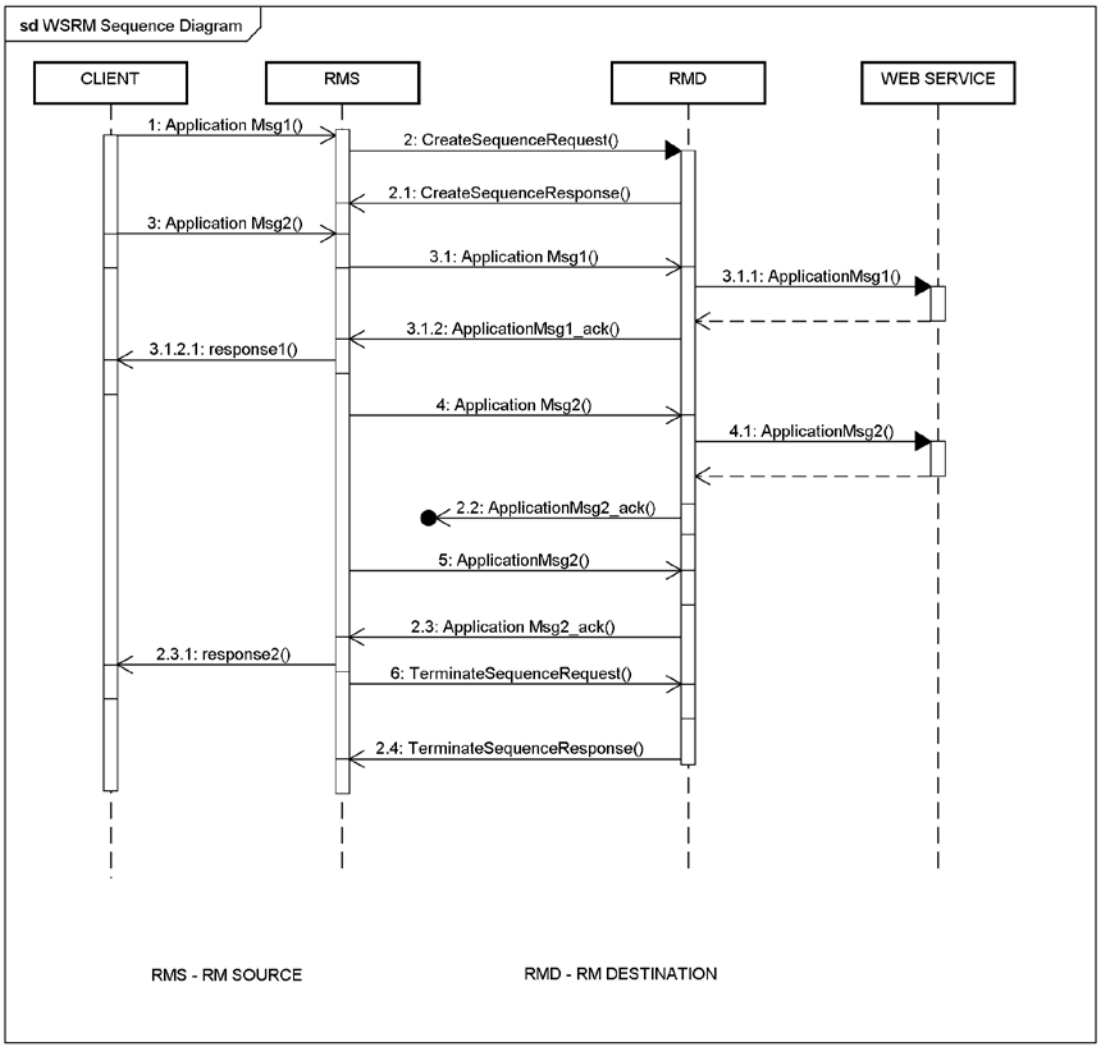


Figure 3. Sequence Diagram of Apache Sandesha2 Functionality.

1.2.3 Insufficiency of the WSRM Specifications Alone

The advent of reliable messaging specifications is considered to be a very good starting point to enhance the reliable interactions of the web services and is widely supported by many commercial and open source frameworks[24][25]. However Web services reliable messaging specifications turns to be inadequate for many mission critical web services to meet reliable needs.

The following points postulate the inadequacy of WSRM

1. WSRM fails to guarantee the high availability of web services which requires space redundancy and is carried out by replicating web services [24].
2. Considering the distributed and untrusted environment in which they often operate, there are legitimate concerns on the security of the web services. Because if a Web service is compromised by an adversary, it may be rendered to provide false/invalid information to the clients [24].

In order to overcome the above mentioned inadequate factors of WSRM we proposed and developed a light weight fault tolerance framework by extending WSRM specifications implementation (Sandesha2 module) to provide it with fault tolerance capability, by replicating the web services to ensure high availability.

1.3 Fault Tolerance

As web services is being adopted majorly in business world to exchange their critical information over web, the dependability of major business modules on web services for critical data exchange is increasing rapidly. Dependability that often varies directly with the high time availability and reliability of the system, forces the researchers and programmers to concentrate on the fault tolerance capability of the systems, using which the availability can be increased. Since the main essence of fault tolerance is to detect errors before hand and subsequently enable the system to work either partially or with low performance at the time of errors, caused due to software or hardware issues.

In a general perspective, the faults and errors may be either directly through the hardware and software failure of the system or through the network overhead. To be specific, software faults are the faults that occur due to the bugs in the software and malfunctioning of the software. Hardware faults occur due to the incompatibility of the hardware components, system or network failures and malfunctioning of the corresponding drivers. The network overhead is generally the congestion or loss of the message packets over the network due to protocol malfunctioning or unexpected data growth.

The fault tolerance capability though provides high availability of a given system, there are considerable amount of challenges ahead of it. First of all, its means [27] are difficult as they are included with the additional system complexity by adding a considerable amount of extra code from the development side, by foreseeing the possible faults and errors. Secondly, it is considered costly process as it always indulges in extra amount of resources including the software and hardware considering the redundancy technique it adopts.

Though making a system highly available through the mechanism of fault tolerance is relatively a costly process, the dependency and increase of many critical business transactions over web services implies the need for their high availability, has suppressed the cost factor incurred with it. The scope and capability of fault tolerance has further advanced with the advent of the new concepts like hot swapping (a way to swap computers hardware peripherals while the system is up and running) and single point

tolerance (to handle hardware fault tolerance).

1.4 Thesis Organization

Our thesis report organization from now on, goes in the following manner. Chapter II deals with a basic introduction of the LFT framework and the system models followed by the related work. Chapter III postulates the consensus problem in the set of asynchronous distributed systems and then the Paxos algorithm for solving consensus followed by the replication algorithm that we adapted from Paxos algorithm. Chapter IV explains the architecture of apache Sandesha2 (open source implementation of WSRM specifications) and the changes introduced by us to Sandesha2 followed by the in-depth client side and server side architecture of our framework. Chapter V deals with the implementation procedure like the server details and the test bed environment followed by the measurement results.

CHAPTER II

LIGHT WEIGHT FAULT TOLERANCE FRAMEWORK

2.1 Framework Introduction

Most of the web systems offer their critical services through web based technologies and most often using web services. As many critical transactions are expected to occur in the backend of these services at the server and transactional level reliability is also given equal importance. So the capability for the systems to reconfigure themselves and to function with or without a very minute downtime in chance of component failure, subsequently providing high availability is considered critical. However, designing and developing such an efficient FT system is considered to be a complex move.

Though there exists many fault tolerance frameworks readily available claiming themselves to provide high availability for web services based transactional systems, they were developed either by extending the fault tolerance frameworks of older generation distributed computing platforms such as FT-CORBA [20], or developed completely ignoring the design specifications of the web services. As argued by many of the researchers Web services are totally different when compared to older generation distributed frameworks [4, 28]. Web services is designed for web based computing over the internet, and it mainly adopts a message based approach for maximum interoperability, whereas the older technologies are not developed for internet and they

primarily focus on the Application programming interface (API) based interactions [24].

Web services are designed and developed to provide an inter operability between different software applications, running on a variety of platforms and/or frameworks [web services doc], and are very much ahead of the previously available distributed frameworks. So a fault tolerance framework, aimed to promise the high availability of web services should also consider the design principles of web services. The previously developed FT-CORBA standard [20], considered one of the major outcomes of fault tolerance research for CORBA, is believed to be a heavy weight framework for CORBA applications itself, same incase if extended to web services.

To overcome all these issues we focused mainly on providing a light weight fault tolerance framework by being bided to the designed principles of the web services. Our framework does not use any complex or proprietary protocol to exchange the messages between the server machine and the backup replica machines and between the client and server. The entire message format with which the conversation takes place within the system is mentioned and exposed through WSDL document which is publicly visible to users and is kept very much close to web service regular service definition standards.

Our framework provides a dynamic switching mechanism between the replication mode functionality of fault tolerance framework to simple reliable specification (WSRM specifications implementation) framework implement. A web service operating in our framework would be made fault tolerant so as to function reliably by using replication

strategy when needed, and switches back to regular reliable specification implementation framework automatically if made to work with a single server replica at any given point of time, and there would not be any extra overhead incurred while this switching of framework goes on back and forth.

Moreover, compared to the complex strategies that are usually incurred while configuring the FT-CORBA [20] framework, our framework's configuration can be done and managed through a simple regular property file. The necessary information that turns to be critical for the functionality of the framework, like the number of server replicas that should function and the logging configuration could be easily configured and altered through a simple property file.

The other works [5, 7, 9, 10, 12, 13, 15, 23], have either made use of checkpoint or replay mechanisms of fault tolerance or they have used totally different strategies. Some of them even ignored the consistency factor achievement while striving to determine failure occurrence in the system, by operating on internet which is an asynchronous system.

2.2 System Models

As our framework deals with web services and clients that interact with each other mostly over internet, which is very much prone to network delays and time delays, we considered an asynchronous distributed system model. We followed some assumptions to maintain the integrity of the messages being exchanged between the client and the server

replicas and also between the server replicas themselves. To safe guard the liveliness of the algorithm and to avoid indefinite waiting times that occur in asynchronous distributed models we proposed and followed a feasible asymptotic upper bound value to ensure some synchrony. The upper bound value is explored and set dynamically and generally doubled when a new view change state occurs.

In order to maintain the safety and liveliness of the algorithm, we assumed a crash fault model while defining our framework. According to which if a web service goes down due to any hardware or software fault or error, it's assumed that it completely stops emitting messages. By this assumption we make sure that not only the server and its replicas but also the clients would not function maliciously at any given point of time throughout the algorithm implementation.

We even considered that there might be some transient network failures in the implementation of the algorithm, but which could be eventually repaired and stored back, but does not allow network partition as consequence. To control the overheads that are generally incurred by asynchronous distributed systems and the non deterministic nature of the web services (for which they are generally prone to) we followed a state machine based approach, which provides us a deterministic environment.

Some assumptions were even made upon the available number of replicas like the atmost possible faulty replica system at any given point of time for the safe proceedings of the algorithm. To be more specific we assumed that if 2f+1 replicas are available, then

at most f replicas could be faulty at any given point of time for smooth functioning of the algorithm. A similar strategy like in [11] is used to determine the unique id's of the available replicas i.e., i value varies between 0 to 2f.

Our implementation proceeds forward in terms of view states. A view is a fixed time period in which a total cycle of client request being served at the server including the primary and backup consistency achievement. In-detail explanation of view feature functionality is out of scope for my work. So for a given view, one system among the replicas would be chosen to function as a primary and remaining automatically functions as backup replicas. A specific equation is followed to determine the primary replica among the available replicas, like the replica with id 0 calculated satisfying the equation i = v mod (2f+1) would serve as the primary for given view v. The value of v starts from 0. For every view change initiated by the view change timer, a new primary is selected using the condition given and the view number is incremented by 1 and there by implementation of the algorithm is carried forward.

2.3 Related Work

Though there are numerous works which have been proposed to provide the solutions for high availability of web services, two of them namely Thema [19] and WS-Replication [16], are more closely related to the framework provided by us, since they ensure to provide strong replica consistency for Web Services.

Thema[19], claims to provide a Byzantine fault tolerance framework for Web

services. This is very much similar to our framework, by following a consensus based approach to achieve consistency among replicas. The main drawback of this work is that it uses an adopter to interface with an existing algorithm [11] implementation, which carries out the message multicast using UDP multicast instead of the standard SOAP/HTTP transport protocol. This is considered to be inconsistent with design principles of Web services and experiences a considerable amount of performance degradation. Moreover, it uses a much weaker fault tolerant model compared to the one provided by us.

WS-Replication [16], is very much close to our work. It maintains consistency among the replicated web services by total ordering the incoming requests to all the available replicas. Though the built and implementation of the client and server models are done by following the web services design principles, the transportation of messages is done by using a proprietary transport system called JGroup communication protocol[18]. Though the JGroup transport protocol supports the SOAP over HTTP, the overall performance is reported to be low. Serialization is considered to be an option to enhance performance, but its usage would violate the design principles of web services. Moreover, the usage of proprietary protocols affects the interoperability of the system, as it introduces language dependency. Even from an implementation perspective WS-Replication uses a separate proxy and dispatcher service to capture the requests from the clients and JGroup messages from replicas [25] that has a considerable amount of degradation in performance. Whereas, in our framework we greatly avoided any kind of extra overhead and performance degradation due to message transport by completely utilizing the

transport protocols proposed by web services design principles, thereby ensuring to provide interoperability.

CHAPTER III

REPLICATION ALGORITHM

In our LFT framework we could provide fault tolerance capability for web services besides the reliable exchange of messages, by using the replication mechanism. We replicated the web services among the replicas and thus guaranteed high availability of the overall system. Though replication mechanism of fault tolerance is a well known and accepted technique, its implementation among distributed systems is considered trivial, due to the hidden problem of solving consensus among them.

3.1 Consensus Nature

Consensus is a well known problem with replication mechanism of providing fault tolerance. In brief it is to obtain consistency among the available replicas by making them agree on a virtual contract. In distributed systems there is a wide chance for the systems to change their behavior dynamically due to factors like up and down of the replicas from the network i.e., the machine or system is subjected to go down at any moment from the network at any point of time due to hardware failures and may even rejoin the network once it is repaired and stored back. So in any of these situations it is necessary for the newly rejoined systems to restore and make themselves cope up with current running systems state and maintaining the consistency across the network.

One best followed approach to attain consistency among a group of asynchronous systems is forcing them to agree on a virtual contract by implementing an instructor-listener mechanism i.e., one system among the group is chosen randomly as a leader and is engaged to control all the other systems, thereby achieving consistency among the whole group of systems on the network. But the critical part here is to reach consensus, in an optimal manner among the available systems.

Many protocols have been proposed to solve consensus among a group of processes, of which Paxos algorithm [22] is considered efficient due to its scalability to any number of processes in a network.

3.2 Paxos Algorithm

Paxos algorithm is one of the renowned ways to solve consensus in a network of unreliable processors. Paxos algorithm ensures to accept and choose a single value among a group of proposers, eventually ensures all the available systems to learn it. All the processors in the network are classified into three roles by Paxos algorithm, namely proposers, learners and acceptors. A single processor may function in a single or more roles at a given point of time.

3.2.1 Processors Roles in Detail

3.2.1.1 Proposer The system functioning in this role would form a proposal value by biding itself to the safety rules of the algorithm. The proposal thus proposed would be transmitted to all other acceptor systems over the network so as to establish a consensus

among them. Mean while there is a chance for other systems present on the same network to form a proposal of its own and start propagating them over the network, thereby affecting the liveliness of the algorithm. In-order to avoid these conflicting situations the algorithm has proposed liveliness rules, like only a proposer chosen as a leader could propose a proposal at any given point of time.

3.2.1.1.1 Leader The special case among the proposers to ensure the liveliness and safety of the algorithm would form a proposal message with the chosen proposal value. Thus formed proposal is transmitted over the network to all other machines which are expected to respond in acceptor role. So at any given point of time the algorithm ensures a single leader to safe guard the smooth functionality of the algorithm.

3.2.1.2 Acceptor The processor's phase is considered to be a fault-tolerant memory of the algorithm. It is capable of receiving proposals proposed by a leader and storing them in their respective logs or memory for further usage. The Acceptor may either discard or respond back to the proposal received from leader with a promise response and discards other previously received less valued proposals. The acceptor would even serve as a member of the quorum, whose fulfillment requirement is used to initiate the phase change of the algorithm.

3.2.1.3 Learner Processors of this role would serve as replication factors. Once a value is accepted upon by quorum of acceptor's i.e., a value is chosen upon according to the algorithm norms, these learners are supposed to learn the value eventually.

Paxos algorithm [22] proposes and governs some of the following safety and liveliness rules for the smooth functionality of the algorithm.

Safety Rules:

- Only proposed values can be learned [22].
- At most one value can be learned(i.e., two different learners cannot learn different values) [22].

Liveliness:

- If value has been proposed, then eventually learner L will learn some value(if sufficient processors remain non-faulty) [22].

3.2.2 Paxos - Phase Wise Functionality

The algorithm's initial phase is termed as prepare phase in which a leader proposer (from here termed as leader) is chosen from a group of available proposer's by strictly following the safety rules of the algorithm.

3.2.2.1 Prepare Phase

Proposers End In this phase the leader (a special case of proposers) would select a value N, forms a proposal and propagates the proposal in the form of a proper encoded prepare message to quorum of acceptor's.

Acceptor's End Once the acceptor receives the proposal from the prepare message with value N, it would validate the current value N against the previous accepted proposal values (if any). After the acceptor makes sure that the current proposal value N is greater

than the previously received proposal values it would promise back the proposer with a prepare response that it would not accept any other proposal with value less than N and also the previously accepted upon value and the current value N.

3.2.2.2 Accept Phase

The prepare responses posted by the acceptors are processed at the leader and once they form quorum, the leader would move on to the next phase of the algorithm i.e., Accept Phase.

Proposer's End The leader would then form an accept request with the proposal value N (promised to be agreed by acceptors) and propagates the accept request to all the acceptors.

Acceptor's End When an accept request is received at the acceptor it would check back with its log files about its promise for the proposal number and if it finds to be the expected Accept request it would respond back to the leader about its acceptance on the value N and also updates its log files about the value N acceptance.

3.2.2.3 Commit Phase

So at the leader's end once a value is accepted upon by the quorum by meeting quorum requirements with accept responses sent by acceptors including the accept request sent by the leader itself, the leader would initiate the commit phase.

Proposer's End The leader would propose a commit request including the value upon which the acceptors accepted upon and propagates the request to all the acceptors for

their agreement.

Acceptor's End When a commit request is received at the acceptors end they would check back with its log files to find the promise corresponding to the value contained by the commit request. If they could successfully find the agreement pertaining to the present commit requests value, they would propose a commit response to respond back to the proposer about the total acceptance of the value and updates its log files with the agreement message they are about to propagate and removes all other previously stored agreements.

Back at the proposer (leader) it would collect back all the commit responses thus generated by acceptors and checks back with its proposed commit request (stored in database), to verify integrity of the value. If the value matches, it marks itself as the value proposed by it has been accepted and committed by all the available machines on the network and there by triggers learners to learn the value committed eventually.

3.2.2.4 Learner Phase

Once the commit responses(on value N) from the acceptor's form a quorum at the leader, the leader would update itself and issue a commit message to all the learners about the acceptance on value N, so that they could eventually learn the value chosen.

3.3 Replication Algorithm

The replication algorithm in our framework has been adapted from the BFT algorithm by Castro and Liskov [11]. Considering the complexity of the original algorithm, we have

greatly simplified it and implemented in our framework.

Of the three phases of the original Paxos algorithm namely Prepare, Accept and Commit phase we just adopted the Accept and the Commit phase omitting the Prepare phase in our framework, to attain consistency among the replicas. According to the Paxos algorithm way of solving consensus among a group of unreliable systems, the Prepare phase is mainly used to determine and agree upon one leader proposer out of all the available proposers and later the chosen proposer would lead the other two phases. But in our framework, as we are following a fixed way to define the primary among all the available replicas for each view i.e., by considering the replica with id value 0 determined from the equation i = v mod (2f+1) is supposed to serve as primary and thereby leader-proposer, we omitted the Prepare phase of the original algorithm. This not only reduces the number of control messages exchanged between the replicas but also helped us to further improve the performance.

Normal Operation

The operating phase of the algorithm starts when a client sends a request targeting one of the replicated web service (most commonly the web service deployed on the primary replica with id 0, of the given view). The request thus sent by the client is generally of the form <REQUEST, s, m, o>, where s being the unique sequence id (defined by WSRM specifications), m being the message number within sequence s and o being the action on the web service that is targeted to be invoked.

Note, that the real messages that are exchanged between the client and the server or between primary and the backup replicas is proper xml messages, formed and documented according to the rules of SOAP and exchanged on the network using an appropriate protocol like http.

The request, sent by the client aiming at a web service, is replicated and sent to all the replicated web services. At the primary replica when the request sent by the client is received, it would first validate the request to check if it is a duplicate one, and if so the request is discarded and an appropriate response is generated and sent back to the client. If the request is not a duplicate request then it is stored in the appropriate local data structure at primary from where further processing is invoked. At the back up replica if the request is validated to be a duplicated one then it is simply discarded without any response generation back to the client for efficiency reason.

At the primary replica, which is the fixed leader in our algorithm, would initiate the consistency achievement phase of our framework by triggering the control message exchange with back up replicas, as soon as a client request is seen in the appropriate data structure, used to store client requests before processing. The primary replica would start the process by proposing an ACCEPT request, as we just adopted only two phases of original Paxos algorithm due to fixed leader, primary replica in our case.

The ACCEPT request proposed by the primary replica would be of the form <Accept, v, n, s, m>, where v being the current view in which algorithm is operating, n being the global sequence number assigned by the primary for the application request message

identified by s and m.

When the primary replica (replica with id 0), is ready to propagate the formed Accept request to all the backups, it would multicast the message to all the backup replicas by utilizing the transport mechanism, implemented as part of the framework. The request that is intended for it is not sent to the network instead it is stored in its own local data structure. On the backup side, when it receives an ACCEPT request it would store the request in the local data structure and prepares a corresponding ACCEPT response to propagate back to primary, to indicate its agreement on the ACCEPT request. The ACCEPT response is of the form <ACCEPT_ACK, v, n>, where v being the current view and n being the global sequence number.

At the backup replicas there is a possibility of the ACCEPT request reaching much before the client request reaches. But this does not disturb the sequence channel of backup and the client, so it keep on receiving the client requests. If at all there is any timeout occurrence due to the premature timeout at the backup, the sequence is again established between the backup and the client and backup requests primary to send any missed requests and makes sure to be consistent with the primary. The whole process doesn't affect the backup replica to accept the ACCEPT request from the primary replica.

When the primary replica receives the ACCEPT response from backup replicas, the response is validated and verified to check the view number and sequence number. If it is same as the view number and sequence number of the request it sent, it logs and stores the response in its local data structure. When the primary receives f+1 messages i.e., f

ACCEPT responses from backup replicas and the ACCEPT request sent by it, the quorum requirement is considered to be reached, there by finishing the ACCEPT phase.

Once the ACCEPT phase is finished successfully at the primary replica, it would propose a COMMIT request of the form <COMMIT_REQ, v, n>, where the v being the view number and the n being the global sequence number on which accept phase has been completed. Once the COMMIT request is formed at the primary replica, it multicasts the message to all the available backup replicas and once again the message for itself is just stored in the local data structure instead of propagating on the network.

At the backup replicas, once they receive the COMMIT request, they would check with its log files to confirm the prior acceptance on the sequence number contained by the COMMIT request and if it finds one, a corresponding COMMIT response would be generated of the form <COMMIT_ACK, v, n> where v being the view number and n being the global sequence number on which acceptance is made. Thus generated COMMIT response is propagated back to the primary replica. In case, the backup fails to find a matching agreement on the received global sequence number, it just simply discards the COMMIT request.

When the primary replica receives the COMMIT response sent by the backup replicas, it validates it for the corresponding view number and sequence number and stores it in a local data structure for further usage. As soon as the primary receives f+1 COMMIT control messages including the COMMIT request sent by it, it marks the message status

bean about the completion of commit phase and proceeds with the total ordering of the application messages sent by the client.

A batching strategy is followed while total ordering of the application messages to the web service, according to which the messages though are FIFO within their sequence, their ordering is postponed till the ordering of already existing k batches is finished, where k being a tunable parameter and often set to 1. The batching mechanism usage is adopted to improve the efficiency further.

As soon as the application messages are ordered to the web service, the corresponding results are logged and the replies are sent back to the clients by generating appropriate response messages only at the primary replica. At the backup replicas the replies are logged but they are not sent to the client through network, unless the backup replica becomes one of the primary replicas through view change procedure.

CHAPTER IV

SYSTEM ARCHITECTURE

Our framework is developed by extending Apache Sandesha2, an open source implementation of Web Service reliable messaging specifications over Apache Axis2 [2]. The apache Sandesha2's source code is distributed under the general public license. Our framework implementation is developed by injecting new additional plug-in code in to the original Sandesha2 code. We even wrapped the original components of Sandesha2 with our custom code to make them function and react according to our needs without losing their original capability and functionality.

The next section would provide a brief explanation about the major components of Sandesha2 and later we would elaborate how we modified or replaced them to function according to our needs in our framework.

4.1 Sandesha2 Architecture

The basic components of sandesha2 and their functionalities are as follows.

4.1.1 Sandesha2 Global In Handler Sandesha2 Global In Handler is the handler class which is invoked in the pre-dispatch phase (global phase) of inflow of Sandesha2. Its behavior could be controlled by changing its properties in Sandesha2 modules configuration file. As this handler operates in global phase of the module, each and every

message that inflows into Sandesha2 would surely pass through this handler. To maximize the performance, this handler is provided with some extra functionality besides checking the correctness of the message, that flows in to Sandesha2.

This handler is provided with some of the functions like the detection for the duplicate messages and dropping them and an appropriate reply would be sent back to the client to intimate it about the message drop. It even handles the functionality of detecting the faults, that are likely to be caused due to reliable control messages and informs them back to the client. It even generates the appropriate acknowledgement responses for the dropped application messages and sends them back to the client [1].

4.1.2 Sandesha2 In Handler Sandesha2 In Handler class is added to the reliable phase of the Sandesha2 module. It poses a special set of message processors that facilitate Sandesha2 module to process the incoming message further basing on the type of the message. This handler is invoked only for the messages that target the reliability enabled services. This handler does the further processing of the incoming message, generally after the Sandesha2 global in handler and invokes the corresponding application message processor depending on the type of the message.

4.1.3 Message Processors These are the special set of classes responsible for processing the messages based upon the type of the message. i.e., each message processor is facilitated to handle a particular type of message. Their main work includes the processing of the incoming message and takes the necessary steps to fulfill all the

necessary requirements for an outgoing message.

4.1.4 Sandesha2 Out Handler Sandesha2 Out Handler does the basic processing on the messages that flow out from Sandesha2 module. This handler would handle the basic functionality of generating the internal sequence id and later replaces them with the corresponding sequence id's once obtained from the reliable messaging destination (RMD) through create sequence message pattern. This handler would even send the messages in a separate sequence or in a group under a common sequence id depending on some decision factors, contained by the messages. A fixed pattern is followed both at server side and client side to define the internal sequence id by Sandesha2 module like on client side a combination of to address and sequence key (unique value given by client) is used and at server side the value is derived from the sequence id value of the incoming messages [1].

4.1.5 Sender Thread This is a continuously running thread, mainly iterates over a local data structure of Sandesha2 module, in which the messages that are needed to be transmitted out are stored. The transmission and re-transmission of the messages carried out by this thread is controlled through the properties defined in the configuration file of the Sandesha2 module. Generally this thread re-transmits or resends the message, if no corresponding acknowledgement is received about a message from RMD in a given time period, guarded by a limit specified in Sandesha2 policy document file, and is often defined dynamically.

4.1.6 Invoker Thread Invoker thread is another continuously running thread over a local data structure of Sandesha2 module which is used to store just the invoker beans (application message oriented beans). This thread has the logic incorporated in it for supporting the features offered through WSRM specifications like the delivery assurance. By default this thread is built in to support InOrder delivery assurance defined by WSRM specifications.

4.1.7 Storage Framework Considered as one of the most important part of whole Sandesha2 framework as it just contains the necessary extensible classes' framework but not the actual persistence implementation. It facilitates user to implement any persistent framework implementation of his choice, to persist the reliable messages. The inbuilt persistent framework is well balanced and organized by facilitating to extend only the required bean managers thereby storing only the required messages into underlying database, which highlights its rich support for loosely coupled nature.

4.2 LFT Architecture

4.2.1 Introduction

We implemented our framework by extending the open source implementation of WSRM specifications, Apache Sandesha2. The major additions we did to the framework include the replacement of in-order invoker of Sandesha2 framework with total order invoker, addition of plug-in classes to side route control messages and new message processors that could handle and process the control messages that are expected to be exchanged between the primary replica and backup replicas.

Most of the additions and changes are made on the server side of the Sandesha2 framework. The client is left as it is except the part to replicate the request messages to all the replicas. The sender thread of Sandesha2 was replaced by a multicast sender and the invoker thread is replaced by a total order invoker. The Sandesha2 frameworks Global In Handler is added with some extra plug-in code to handle and separate the messages pertaining to control messages. The framework provided by us is also backward compatible, that is if we don't need the functionality of fault tolerance framework we can easily switch back to the basic WSRM specifications implementation framework dynamically, and to do we don't require redeployment of web services. The changes made to the Sandesha2 components are described in detail in the following sub section.

4.2.1.1 Sandesha2 Global in Handler We have added some plug-in code to the basic Sandesha2 global in handler to detect the control messages that are exchanged between the primary and the backup replicas, to ensure consistency between them. The control messages thus detected are re-routed to newly provided message processors, at which they are processed and necessary steps are taken.

4.2.1.2 Sandesha2 Out Handler This handler of Sandesha2 is basically responsible for processing out going messages and mainly it handles the establishment of sequence channel by creating and handling create sequence messages and termination of the reliable channel by producing terminate sequence message. In our framework we changed the create sequence message handling and there by the original Sandesha2 out

handler, so as to detect the duplicate create sequence messages. In our framework the create sequence messages could be exchanged between the replicas at the server side to form a reliable messaging channel for exchange of control messages. If the create sequence message contains an offer element then it could be a way to detect the duplicate messages, however not all create sequence messages holds the offer element as its existence is specified by client. We overcame this issue by introducing an addition of a UUID string in the create sequence message.

WSRM specifications does not specify how a sequence ID for a newly create sequence should be determined [24]. In Sandesha2, at server side a UUID is generated and used, and if the same strategy is followed the client would accept the UUID from first create sequence response, which would stop client from communicating with other replicas and consistency in UUID is not maintained by replicas. So in our work the create sequence message is altered to handle the UUID generation deterministically.

4.2.1.3 Multicast Sender The actual sender thread component that involves in sending the messages out of Sandesha2 module is replaced by a multicast sender component. The multicast sender component would dynamically manage the mapping between the replicas. To allow the easy mapping, we assumed that every web service needed to be replicated possess two end point address, one for unique individual endpoint and the other for group end point. The application and the higher level components are expected to use the group end point while referring to replicated web service. Once the multicast sender component receives a message with group end point, then it replicates the message with individual end points and multicast them correspondingly.

The multi sender component would keep on polling the sender queue like the original sender component and dynamically adjusts its capability to either multicast or send message normally. The client side architecture also uses this multicast sender thread to multicast the messages to all replicated web services. This sometimes proves to be inefficient considering the geographical distance between the client and services, but would surely improve the robustness and security. By using the multicast message sender strategy we are not only encapsulating the system information from the clients but also protecting the primary replica from adverse effects, since the clients would not be aware of the primary serving replica. The encapsulation of the information of the replicas and web services from clients would increase the robustness of the whole system.

4.2.1.4 Total Order Invoker Sandesha2 framework offers an in-order delivery of the messages to the web service through an invoker thread, whose properties are controllable through configuration file. The invoker thread regularly polls the In Messages Queue, in which the received application messages are stored. In our framework we replaced the Sandesha2's invoker with a total order invoker thread. The message is considered ready for ordering if it is in-order within its sequence and all the messages prior to it in the given sequence are ordered or being ordered. Once a message is considered ready for ordering, the total order manager is intimated about the ordering of the message and is stored in a local data structure.

The total order invoker which regularly polls total order manager for any new ordering

messages would now order the message by executing it to the web service through the Axis2 module [2]. In our framework only the primary replica would be ordering the message to the web service.

4.2.1.5 Total Ordering Manager The total ordering manager is the new component added by the framework to impose total ordering over the messages by providing a separate status tracking object, total ordering bean. The total ordering manager would initially trigger the exchange of the control messages between the replicas to ensure consistency between them and would then allow the total ordering of the application messages sent by client. The total manager would initiate the replication algorithm, when an application message in order within its sequence is received. A total ordering bean is also initiated to maintain the status as soon as a first control message is proposed or received by the replica.

Moreover the batching mechanism is implemented in total ordering manager to improve the efficiency and performance, through which the ordering of the messages, that are ready for ordering by being in-order within its sequence, is postponed till the k batches of messages before it are ordered or being ordered, where k being the tunable parameter and is often set to 1.

4.2.1.6 Replication Engine Replication engine is key addition to the framework through which the whole replication algorithm is executed. It possess its own log files to create and track back the store points and also the addresses of the available replicas is made

available for the multicast sender component, to multicast the group end point oriented messages.

The client side architecture used in LFT as shown in Figure 4, is almost left unchanged from original sandesha2 client architecture, except for the multicasting capability of the sender thread which is used for the messages that target the group end point, so as to multicast the application requests to all the replicated web services. The formation of the replication of the application request and process of multicast is handled in low level architecture so the whole process is abstracted from the actual Sandesha2 client code.

4.2.2 LFT Client Side Architecture

The high level client side architecture of Light Weight Fault Tolerance framework is shown in Figure 4. At the client side no changes are made to the system i.e., the Sandesha2 module engaged on the client side would always function in normal mode. The client prepares the application message with appropriate information and triggers it on the network to transmit it to the server. The Sandesha2 module which is engaged on the client side would receive the application message and passes the message through its various components for further processing. The following sub section elaborates an in-detail message transmission that happens at the Sandesha2 module engaged at client side.

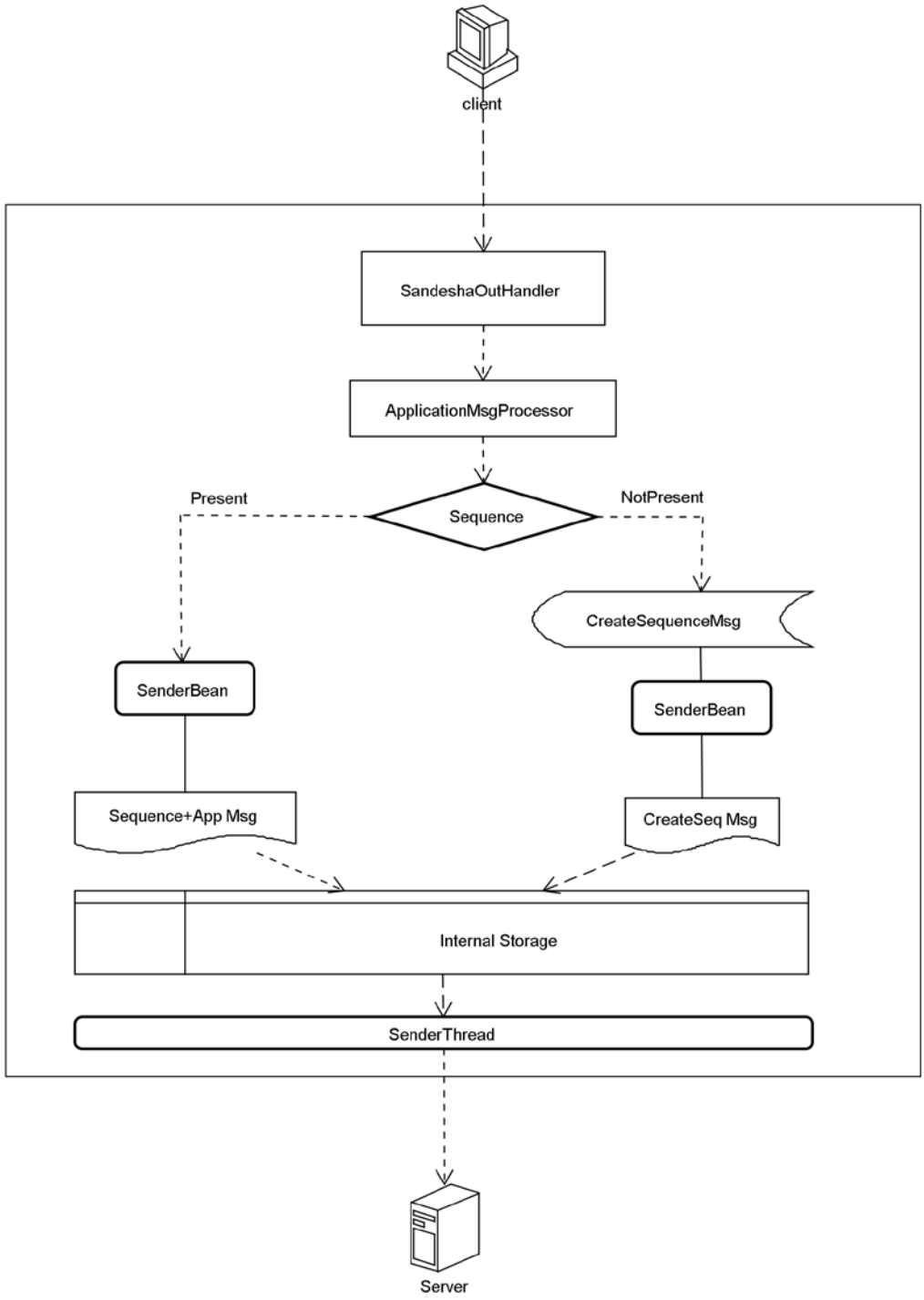


Figure 4. LFT Client Side Architecture.

Sandesha2's Global-In Handler being in pre dispatch would be the first component to

receive the application message and eventually passes the message to Sandesha2-Out Handler. Sandesha2-Out Handler would check for a sequence ID to transmit the message to the appropriate server, which has to be already present if the message is not a first message. If Sandesha2 out Handler detects the application message to be a first message to the appropriate server, it would pause the application message by placing it in sender queue with send status false and generates a CREATE SEQUENCE REQUEST control message to establish a sequence ID with the corresponding server (which is a mandatory requirement according to WS reliable messaging specifications implementation). Then the control message is placed in the sender queue of Sandesha2 with send status true, which is later verified and transmitted by sender thread to appropriate destination.

After a while the Sandesha2 module on client side would receive a CREATE SEQUENCE RESPONSE control message from the server with a Sequence ID, which is collected and saved for further usage. Thus obtained sequence ID is used by Sandesha2 module to invoke all the paused application messages pertaining to the respective server and update them with sequence ID. The application messages then are placed in the sender queue, marking their send status to true, indicating they are ready for transmission to the destination by the sender thread. The sender thread would then transport those messages to appropriate destinations by using Axis2 engine.

4.2.3 LFT Server Side Architecture

The high level server side architecture of the Light Weight Fault Tolerance framework is shown in Figure 5. Some additions and replacements are made to original Sandesha2

module as shown in Figure 5, that operates on the server side, which include addition of some new components like Replication Engine, Control Message Processor, Total Ordering Manager, Total Ordering Bean etc and replacements like In-order Invoker

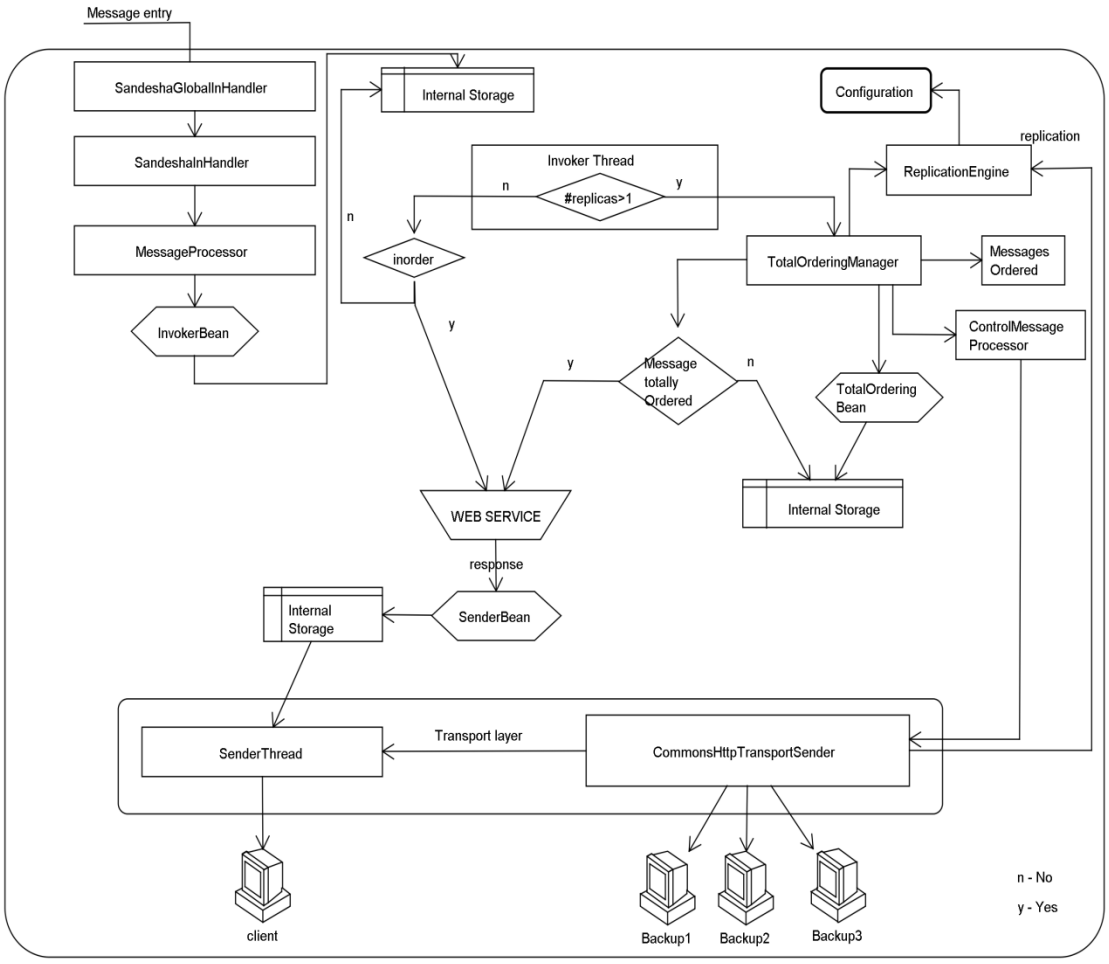


Figure 5. LFT Server Side Architecture.

to Total Ordering Invoker for performing total ordering of messages, sender thread to multi-cast sender to propagate messages to all the backup server replicas by fetching corresponding addresses from replication engine.

Though the web service is replicated along with the replication of primary server, the

ordering of application messages (received at the server) to the web service is done only at primary server replica. The backup replicas just act as backups for control message storages, by being synchronous with primary. Before starting the actual ordering of the messages at primary, it exchanges couple of control messages with the available backup replicas to ensure their consistency with it.

4.2.3.1 Exclusion of Prepare Phase

Of all the available server replicas the primary server replica is chosen to be the leader and all other backup replicas are treated as acceptors. As we are considering a fixed leader, we considered to skip the Prepare Phase and continue directly with Accept and then with the commit phase.

4.2.3.2 Accept Phase As soon as the primary replica gets an application request from the client, it is paused and stored in a local data structure, and the total ordering manager component of primary replica would assign a global sequence number to the request and a corresponding total ordering bean is created. The primary replica would make sure that each and every total ordering bean thus formed, goes through whole control phase process before it is totally ordered, to maintain consistency.

At the primary replica once the total ordering bean is formed, the control message exchange phase between primary and backup replica is triggered. An accept request is generated at the primary replica and sent to the Sandesha2's multicast sender component which would multicast only the required messages basing on the SOAP action property

of the message. Multicast sender component would obtain all the required backup replicas addresses from the replication engine.

At the backup replica once an Accept request is obtained it is processed by the control message processor component. A new total ordering bean is formed at each backup corresponding to global sequence id. An Accept Response control message is then generated at the backup which includes the global sequence Id, and sent back to the primary replica, indicating its acceptance on the accept request.

The primary replica would collect all the Accept responses pertaining to a global sequence Id and checks to forms a quorum of Accept messages including the Accept request sent by it. Once the quorum is reached by the accept control messages, the accept phase is counted to be done, appropriate changes are marked to total ordering bean and the commit phase is initiated by the primary replica.

4.2.3.3 Commit Phase The primary replica would then generate a Commit request including the global sequence Id on which quorum of acceptance is reached by accept control messages. Then once again a similar procedure is followed at primary replica's multicast sender component to multicast the commit request to all the available backup replicas.

At the backup replica once the commit request is obtained it is validated by the Sandesha2 global handler and forwarded to control message processor component for

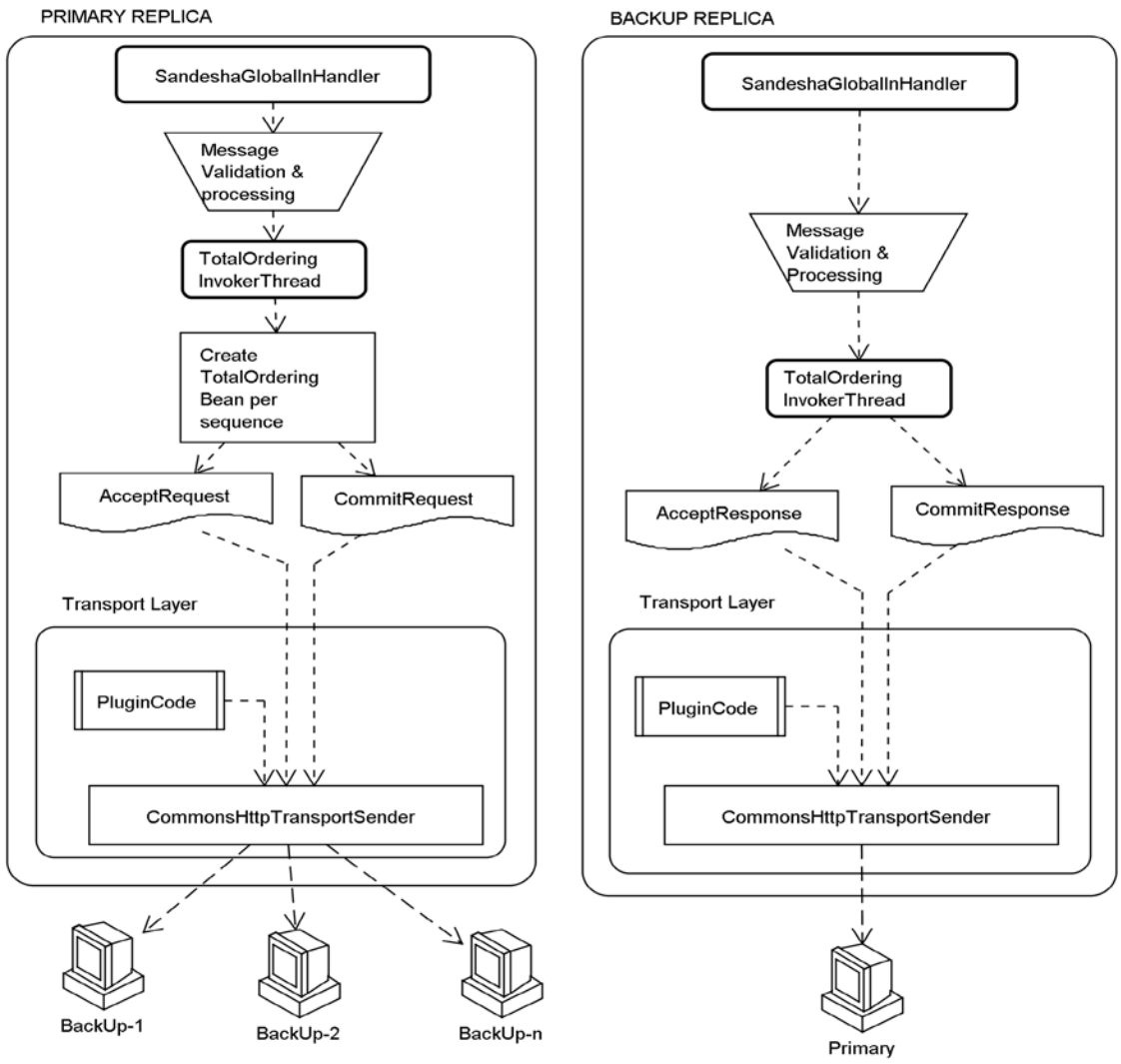
further processing. The corresponding total order bean at backup identified by the global sequence id would be updated regarding the commit request and a corresponding commit response is generated by the backup and transmitted back to the primary replica.

The primary replica would collect all the commit responses sent from backups and checks to complete a quorum on commit control messages along with the commit request sent by it. Once the quorum on commit control messages is completed the corresponding total ordering bean is updated. This ends the commit phase for a total ordering bean and the total ordering manager would move this bean into next phase i.e., to total order the messages.

The completion of commit phase for a total ordering bean would make it eligible for total ordering of its messages. Total Order Invoker component would take care of ordering the messages to the web service by utilizing Axis engine. Here we also introduced a batch mechanism to improve the performance, by which we would postpone the total ordering of the messages till an ordering of already existing k batches is completed [25]. Here k is a tunable factor and by default set to 1.

The message transmission between the various internal components of the primary and backup replica is shown in Figure 6. The primary and backup replicas though bear similar configurations, are distinguished with the properties mentioned through the simple properties file. They react only to the corresponding set of messages that they are configured to react for. The primary replica is configured to propose the ACCEPT and

COMMIT request and propagate them to the backup, but it's only at backup replicas these request messages are processed. In the same way ACCEPT and COMMIT responses are proposed at the backup replicas but they are processed only at the primary replica.



(a) (b)

Figure 6. Message transmission at replicas. (a) Message transmission between internal components at primary replica. (b) Message transmission between internal components at backup replica.

CHAPTER V

PERFORMANCE EVALUATION

We implemented the testing of our framework on a test bed comprising of 12 Dell SC440 servers, which communicate with each other over a 100 Mbps Ethernet channel. Each server in the network consists of a Pentium D 2.8 GHz processor and 1 GB ram of memory. A simple echo test client is used to measure the performance of our framework. The client would send the request targeting the replicated web services and waits to receive the corresponding response. The client request is a proper XML document written according to the SOAP standards, using AXIOM (Axis XML Object Model) model language [3]. The replicated web service would generate almost an identical XML document with SOAP standards, after parsing and processing the received request.

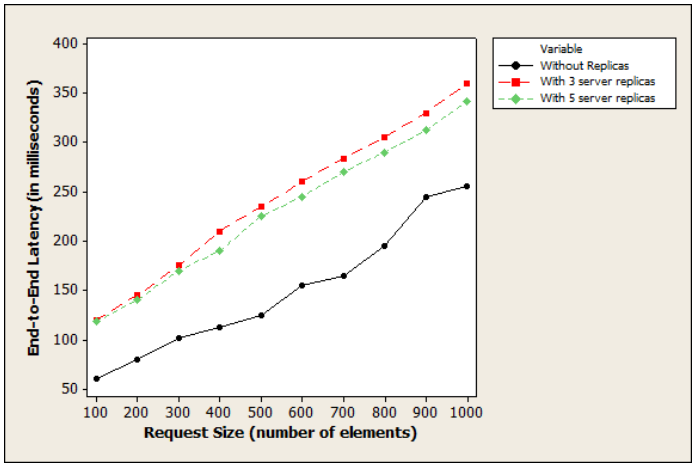
Here we are focusing mainly on the runtime overheads of our replication algorithm while going through normal operation phase. When the primary replica goes down, a considerable time lapse is seen on the client side. Especially, when the client is waiting on a replicated web service to order the request sent by it because a view change is expected to occur and a control message exchange is triggered between the primary and the backup replicas to ensure its consistency in the network. After this process the primary is expected to go on with its normal message ordering phase.

To avoid indefinite wait time, that is possible in asynchronous systems operating over network, a proposed timeout value is also followed. Once the delay is equal to the proposed timeout the algorithm is restarted. The timeout value chosen is generally higher considering the asynchronous message delays. In our experiment we set the timeout value to 2 seconds in LAN environment, but in internet environment it should be relatively higher value. So if a primary replica fails consecutively, a greater time delay is experienced by the client waiting for the response.

A significantly less delay is noticed in case of backup replica down. Though a re-join of backup replica indulges in exchange of control messages to maintain its consistency with other replicas, it happens without any pause to the normal execution of the replication algorithm and avoiding any significant chance of time delay. Moreover only primary replica is expected to order messages to replicated web services, backup replica failure least effects the replication algorithm.

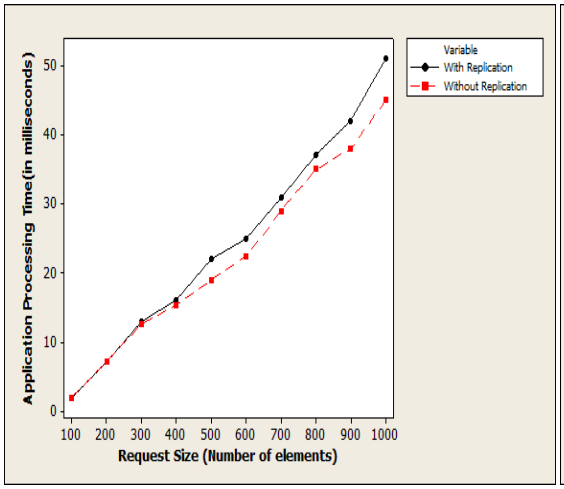
We measured the latency of the framework at client side and measured the throughput and application processing time (in case of heavy load) at the replicated web service. We obtained 1000 samples for each run and analyzed them in detail to graph the measurements. We experimented by varying the number of clients, load on client requests and number of replicas on server side.

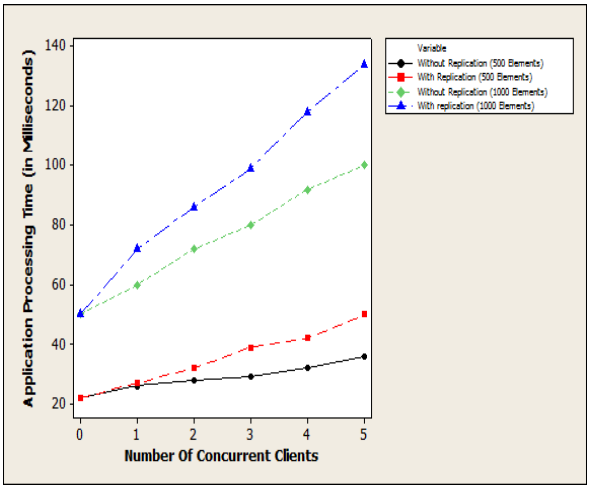
Graphs:



NOE	no replica	3 replicas	5 replicas
100	55	124	126
200	76	148	153
300	106	162	166
400	120	186	210
500	136	218	238
600	156	242	256
700	168	256	276
800	188	280	298
900	232	316	326
1000	244	328	360

(a) (a.1)



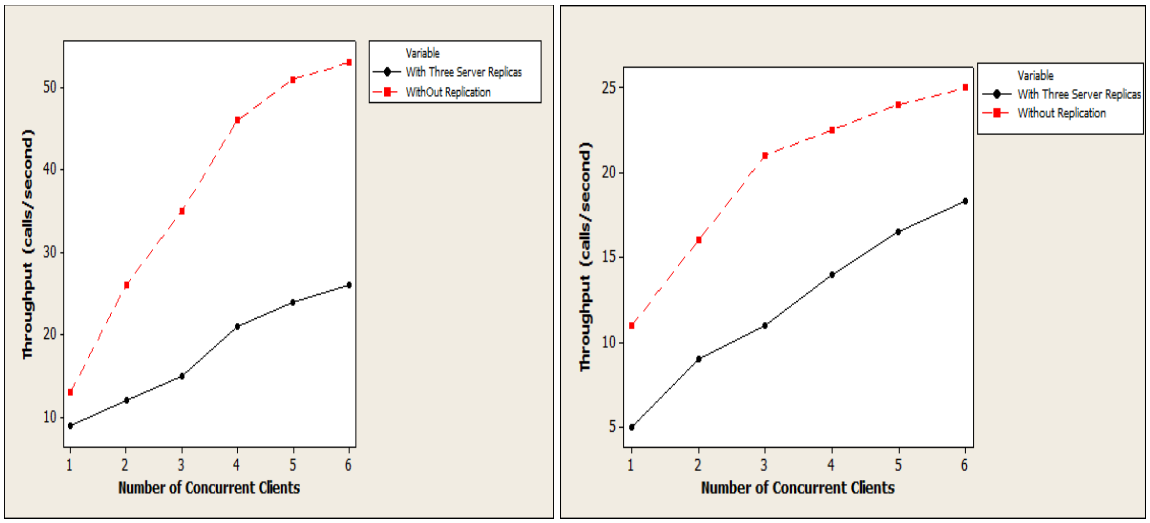


(b) (c)

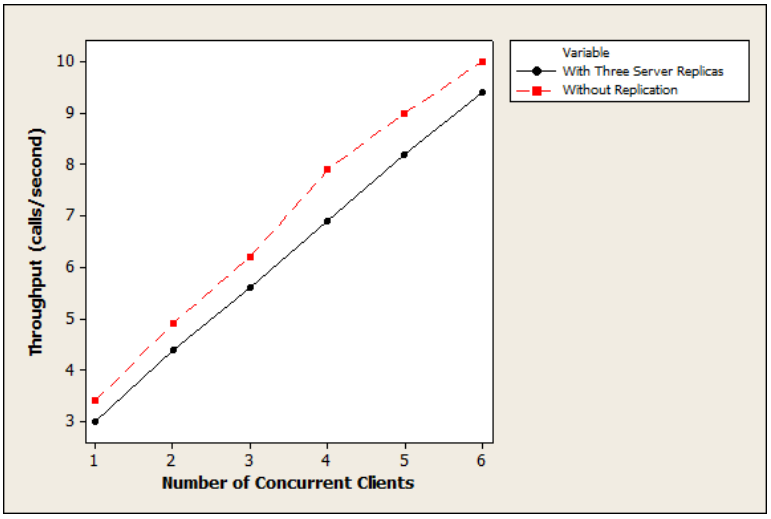
Figure 7. Indicates the latency measurement results & application processing time measurement results. (a) Indicates latency measurement with varied replication degrees of 1, 3 and 5, followed by the corresponding readings in tabulated form in (a.1). (b) Application processing time for requests of different sizes. (c) Application processing time with different number of concurrent clients.

Latency measurement results are shown in Figure 7. While measuring the end-to-end latency we varied the degree of replicas in terms of 1, 3 and 5 count, shown in Figure 7(a). As our framework switches back to WSRM implementation while working with 1 replica without adding any extra overheads, significantly smaller latency values are noted compared to other scenarios. We also provided a tabulated form of measurement results that we noted to determine the end-to-end latency, to clearly indicate that our framework rollbacks to simple implementation of WSRM specifications without producing any kind of extra overhead, incase of no-server replicas scenario, shown in Figure 7(a.1).

A measured value of 60 ms to 100 ms latency is noticed by our replication framework with a replication degree 3 while ranging from smaller to larger requests respectively, as shown in Figure 7(b). Comparatively high latency while working with larger requests is expected due to more likely chances for contention of the messages at the CPU that needs processing (ordering by web services) and the overhead introduced by the replication mechanism. The overhead due to the CPU contention is seen even more in case of concurrent client's usage, as shown in Figure 7(c). A substantial increment in the replication degree i.e., from 3 to 5, has shown a very small change in latency variation, as shown in Figure 7(b) and 7(c).



(a) (b)



(c)

Figure 8. Indicates the throughput measurement results. (a) 100 elements per request. (b) 500 elements per request. (c) 1000 elements per request.

Throughput measurement results are shown in Figure 8. In our experiment we measured the throughput at the replicated web service. From the Figure 8(a), we could infer that a significant amount of degradation in throughput is noticed in case of smaller requests usage especially when replication is enabled and is even more in case of

concurrent clients scenarios. The degradation is considered to be obvious because even with 6 concurrent clients the primary has to send 3 control messages and receive 6 control message responses for ordering 6 application messages. From the Figure 8(b) and (c), we can notice that as the complexity of application messages went up the degradation in through put came down.

Compared to the previous works [16] that reported a noticeable 2/3 throughput degradation when web services were implemented with standard SOAP protocol, our frameworks 50% reduction in throughput is considered optimal.

CHAPTER VI

CONCLUSION

From our work we could successfully develop a light weight fault tolerance framework for web services by totally abiding to the design specifications of web services. We were also successful in producing a backward compatible framework i.e., with an easy switching capability between the replication mode framework (our framework) to the non replication framework (to function as just a WSRM specifications implementation), upon requirement. Moreover, the switching mechanism happens dynamically without producing any overhead, upon availability of the resources and the requirements. We even concentrated to carefully tune the framework to operate with optimal performance, which we highlighted through our measurement results.

REFERENCES

[1] Apache Sandesha2, *http://ws.apache.org/Sandesha2/Sandesha2/*

[2] Apache Axis2, *http://ws.apache.org/axis2/*

[3] Apache Axiom, *http://ws.apache.org/commons/axiom/*

[4] W.Vogels, "Web Services are not distributed objects", IEEE Internet Computing, pp. 59-66, November-December, 2003.

[5]K.Birman, Adding high availability and automatic behavior to Web Services, *Proceedings of the* 26th International Conference on Software Engineering, Scotland, UK, May 2004.

[6]K.Birman, Reliable Distributed Systems: Technologies, Web Services and Applications, Springer, 2005.

[7] A. Erradi, P.Maheshwari, "A broker-based approach for improving Web Services reliability", *Proceedings of the* IEEE International Conference on Web Services, Orlando, Florida, July 2005.

[8] S.Graham et al., Web Services Resource 1.2, OASIS Standard, April 2006.

[9] L.Moser, M.Melliar-Smith, and W.Zhao, "Making Web Services dependable", *Proceedings of the* First International Conference on Availability, Reliability and Security, Vienna University of Technology, Austria, pp. 440-448, April 2006.

[10] G.Dobson, "Using WS-BPEL to implement software fault tolerance for Web Services", *Proceedings of the* 32nd EU-ROMICRO Conference on Software Engineering and Advanced Applications, pp. 126-133, July 2006.

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

November 2002.

[12] P.Chan, M.Lyu, and M.Malek, "Making services fault tolerant", *Lecture Notes in Computer Science*, Vol. 4328, pp, 4361, 2006.

[13] V.Dialani et al., "Transparent fault tolerance for Web Services based architecture", *Lecture Notes in Computer Science*, Vol. 2400, pp. 889-898, 2002.

[14] R.Bilorusets et al., Web Services Reliable Messaging Protocol Specification, February 2005.

[15] C.Fang et al., "Fault tolerant Web Services", *Journal of Systems Architecture*, Vol. 53, pp. 21-38, 2007.

[16] J.Salas et al., "WS-Replication: A framework for highly available web services", *Proceedings of the* 15th International Conference on World Wide Web, Edinburgh, Scotland, pp. 357-366, May 2006.

[17] G.Santos, L.Lung, and C.Montez, "FTWeb: A fault tolerant infrastructure for Web Services", *Proceedings of the* IEEE International Enterprise Computing Conference, Enschede, The Netherlands, pp. 95-105, September 2005.

[18] JGroups: A Toolkit for reliable Multicast Communication. *http://www.jgroups.org*.

[19] M.Merideth et al., "Thema: Byzantine-fault-tolerant middleware for Web Services applications", *Proceedings of the* IEEE Symposium on Reliable Distributed Systems, pp. 131-142, 2005.

[20] Object Management Group. Fault Tolerant CORBA (*final adopted specification*). OMG Technical Committee Document ptc/2000-04-04, April 2000.

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

1982.

[22] L.Lamport, “Paxos made simple”, *ACM SIGACT News* (*Distributed Computing Column*), Vol. 32, No. 4 pp. 18-25, December 2001.

[23] N.Looker, M.Munro, J.Xu, “Increasing Web services dependability trough consensus voting”, *Proceedings of the* 29th Annual International Computer Software and Applications Conference, pp. 66-69, 2005.

[24] W.Zhao, “Design and implementation of a Byzantine Fault Tolerance Framework for Web Services”.

[25] W.Zhao, “A Light Weight Fault Tolerance Framework for Web Services”.

[26] Web Services Architecture Specifications, *http://www.w3.org/TR/ws-arch*.

[27] Abd-El-Malek, M.Ganger, G.R.Goodson, G.R, Reiter, M, Wvlie, J.J., 2005. Fault-scalable Byzantine fault-tolerant services. In: *Proceedings of* ACM Symposium on Operating System Principles.

[28] J. Maurer, “A conversation with Roger Sessions and Terry Coatta”, *ACM Queue*, Vol. 3, No. 7, pp. 16-25, 2005.