QUEUED TRANSACTION PROCESSING WITH WEB SERVICE RELIABLE MESSAGING

A.C. SURIARACHCHI

This Dissertation was submitted to the Department of Computer Science and Engineering of the University of Moratuwa in partial fulfillment of the requirements for the Degree of MSc in Computer Science specializing in Software Architecture

Department of Computer Science and Engineering
University of Moratuwa
February 201 0

96424

ABSTRACT

With the popularity of the distributed business applications, the application data is distributed in various physical storages. However most of the business transactions require to update data stored in more than one storage. hence updating two data storages reliably is a common problem for most of the distributed business applications.

Queued transaction processing is a concept widely used to achieve such a processing model using intermediate queues to transfer messages reliably. In such a system at the client side, both updating the client storage and writing the message to be sent to the client side message queue happens in the same distributed transaction. Similarly at the server side reading the message from the server side queue and updating the sever storage happens in the same distributed transaction. Bur such a system may have interoperability problems if client and server use different types of technologies.

Web services are used to communicate among the heterogeneous systems by passing SOAP messages using standard transport mechanisms like http. Web services can reliably communicate by using WS-Reliable messaging specification(WS-RM). WS-RM uses concepts of Reliable messaging source (RMS) and Reliable messaging destination (RMD) between which it guarantees reliable massage delivery.

By combining these two concepts, we introduce an approach to solve the above mentioned problem in an interoperable manner using WS-RM ..,to communicate between nodes while keeping RMS and RMD as intermediate storages. In our model reliable message delivery happens in three phases. First both updating application client storage and writing message to the RMS happens in the same distributed transaction. Then WS-RM protocol reliably transfers the message to RMD at the server side . Finally- at the server reading the message from the RMD and updating the server storage happens in the same distributed transaction. The middleware software entity that we developed to encapsulate this approach is called Mercury which implements WS-RM protocol.

DECLARATION

"The work included in this report was done by me, and only by me, and the work has not been submitted for any other academic qualification at any institution"

[illegible]

Name: A.C. Suriarachchi (088254P)
Date: 2010.02.26

"I certify that the declaration above by the candidate is true to the best of my knowledge and that this dissertation is acceptable for evaluation for the Degree of M.Sc in Computer Science specializing in Software Architecture"

UOM Verified Signature

Project Supervisor: Dr. Sanjiva Weerawarana
Date: 2010.02.26

ACKNOWLEDGMENTS

I wish to sincerely thank my supervisors Mr. Paul Fremantle and Dr. Sanjiva Weerawarana for providing me the research idea and supervision of my work continuously. They provided me necessary guidance, various levels of requirements and encouragement to fulfill my objective. I would also like to thank Dr Srinath Perera who reviewed my work and provided me valuable feedback. I am also grateful to Prof. Gihan Dias and Dr Sanath Jayasena who worked as the course coordinators, and provided valuable feed back at various levels of the project. I would like to extend my thank to all the academic staff of the University of Moratuwa for the great work they did for us during the course of study.

No student can survive in a university without the help of their fellow students to discuss ideas, share opinions, and to make time spent in the lab and all round enjoyable experience. I would be grateful for all M.Sc 08 colleagues for the corporation given to the successful completion of my project involvements.

I must also be grateful to my parents and brothers for the encouragement they provided to follow the Msc. I wish to express my gratitude to all my colleagues at WSO2 who have enormously helped to learn a lot about web services and distributed systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

WS	Web Service
WS-RM	Web Service Reliable Messaging
RMS	Reliable Messaging Source
RMD	Reliable Messaging Destination
2PC	Two Phase Commit
JTA	Java Transaction API
JTS	Java Transaction Service
SOAP	Simple Object Access Protocol
RPC	Remote Procedure Calls
MOM	Message Oriented Middleware

Chapter 1

Introduction

1.1 Background

Updating two data storages reliably is a widely researched area in distributed computing. Most of the existing solutions follows a queued transaction processing model. In such a model first client writes the message stored in its persistence storage to the request queue within a distributed transaction and server reads the request from the request queue within another distributed transaction. If there is a response to be sent, server writes the response to response queue within the same transaction it read the message and finally client reads the response from the response queue within another distributed transaction. This processing model can operate even with the presence of the node failures due to recovery nature of the distributed transactions. However these systems may not properly inter-operate due to use of many proprietary messaging protocols.

Web services are used to communicate among the heterogeneous systems by passing SOAP messages using standard transport mechanisms like http. Web services can reliably communicate by using WS-Reliable messaging specification(WS-RM). WS-RM introduces concepts of Reliable messaging source (RMS) and Reliable messaging destination (RMD) between which it guarantees reliable message delivery. As a result of this both RMS and RMD can be considered as intermediate queues by using a persistence storage to implement them.

An inter operable reliable message transferring system can be made by combining above two concepts where client writes messages from the persistence storage to RMS within a distributed transaction and server writes the messages from RMD to persistence storage within a distributed transaction. Reliable communication between RMS and RMD is guaranteed by WS-RM.

1.2 Abstract Problem

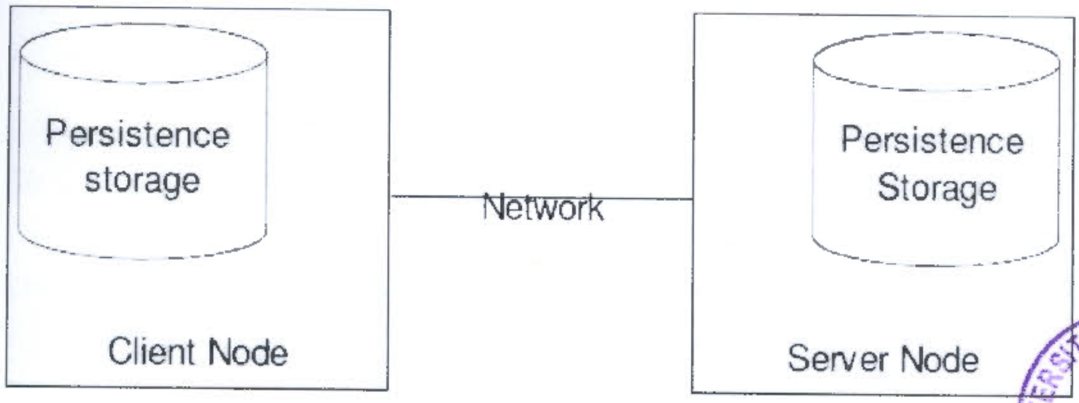


Figure 1-1 Abstract Problem

This project focus on updating client persistence storage and server persistence storage reliably by sending a message in a system shown in Fig 1-1. Assume there are two nodes called client node and server node with persistence storages, connected through a network. How to guarantee both client side and server side storage updates by sending a message from client node to server node with the presence of failures in an inter operable manner?

Message provides the necessary information to update the server persistence storage. The term reliably refers to the exactly one delivery. This means there can be no message losses or duplicate messages. Failures can either be network or node failures. For this work web services and standards are being used as the means of achieving interoperability. Further it is assumed that although it is possible to have network and node failures they recover in finite time and there are no persistence storage failures.

The main goal of this project is to implement a web service reliable messaging middleware which can generally be used in such a situation. Writing a WS-RM implementation from the scratch means a lot of work. Therefore this project aims to re-engineer the existing WSO2 Mercury to solve the above mentioned problem.

WSO2 Mercury is a WS-RM implementation written on top of Axis2 by using a state machine model. However WSO2 Mercury keeps the state of the WS-RM communication in an in memory object model. It achieves the persistence by saving this in memory object model to a persistence storage. Although WSO2 Mercury has successfully implemented the state machine model, its' in memory model described above does not allow it to support user initiated transactions.

Therefore the main objective of this project can be narrowed down to re engineer the existing WSO2 Mercury code to come up with a storage API which support user level transactions. However some of the WS-RM usage scenarios do not require user level transactions and hence it is enough to have an in memory storage model. Therefore above storage API should support simple in memory implementations as well.

1.3 Method of study

Implementing a new storage API directly with the WSO2 Mercury can be complex. Further in such an attempt main focus may not be in the storage API design. Therefore this project first designs the storage API within a simulator. Again the simulator which is used at the time of designing Mercury state machine model can be used for that. Then the new storage API can be implemented in an in memory model with the simulator and can be transferred to the actual Mercury implementation with the necessary refactoring of the Mercury. Finally the storage API can be implemented with a persistence storage and can be tested for distributed transaction scenarios.

1.4 Previous work

As given in the background section this problem has been solved by using intermediate queues. But this project aims to do that using web services and related standards to achieve interoperability.

Apache Sandesha2 which is another WS-RM specification implementation uses such a transactional data store model. However Apache Sandesha2 does not use a state machine model and further a transactional storage to support even an in memory model as well.

1.5 Expected result

In summary this project aims to come up with a storage API with the necessary WSO2 Mercury runtime architecture which supports both user level transactions and simple in memory implementations. To prove this point it expects to have at least two storage API implementations one for simple memory implementation and other for a transactional permanent storage implementation. Further it aims to provide necessary usage scenarios which uses the distributed transactions to achieve end to end reliability.

Chapter 2

Literature Review

Literature review of this projects spans across many areas. One of the obvious areas is the web services standards and related specifications. Web services primarily use SOAP[3] as the messaging format. WS-Addressing[10] provides a way to address end point references in a transport independent way. WS–Reliable messaging specification[8] uses WS-Addressing[10] to correlate the request and response messages.

There are some set of standard protocols and standards to generally support transactions and messaging. 2PC[7] is the widely used protocol to achieve distributed transactions. X/Open promotes standards for many protocols to improve the interoperability. X/Open distributed transaction specification[11] standardize the use of 2PC protocol. JTA/JTS[12][13] provide java specific APIs for distributed transactions.

WS-Transaction specifications which includes WS-Coordination[4], WS-AtomicTransactions[5] and WS-BussinessActivity[6] provides means to achieve distributed transactions using 2PC protocol in an inter-operable way.

Queued transaction processing is used for processing a transaction between a client and an application server asynchronously in a distributed transaction processing environment having at least one transaction queue manager.

IBM has done some work[14] related to this area. This includes their classification of varios ways to integrate the web services and transactions. Httpr is an effort to build a reliable protocol on top of Http.

Finally there have been many researches for message oriented and object oriented transactions. Further these researches have been extended to middleware mediated transactions[15] which combines the above two concepts to achieve better transaction support.

2.1 Web service standards

2.1.1 SOAP

Simple Object Access Protocol (SOAP)[3] is a protocol to exchange information in a decentralized, distributed environments developed by Microsoft and IBM. SOAP can support to enable remote procedure calls (RPC) over HTTP using XML. SOAP protocol specification mainly consists of three parts.

1. SOAP Envelope

SOAP envelope describes what is in the message and how to process it. A SOAP envelope has a required body part which is used to send the actual message, and header parts which can be used to provide the soap envelope processing instructions.

2. Set of encoding rules

There are a set of encoding rules which specify how to encode application-defined data types in to XML format. This is important since SOAP provides an inter operable XML based messaging format.

3. Convention to represent remote procedure calls and responses

SOAP defines a way to encode a RPC invocation request and the response into a SOAP envelope. This is used in RPC type service invocations.

2.1.2 WS-Addressing

Web services can be accessed by sending SOAP messages to their respective endpoints. However the endpoint details may depend on the transport protocol. And also there are some information required by the messaging systems in order to dispatch messages to corresponding processes and correlate them.

Web Services Addressing (WS-Addressing)[10] defines two inter operable constructs that convey information that is typically provided by transport protocols and messaging systems. These constructs normalize this underlying information into a uniform format that can be processed independently of transport or application.

1. Endpoint references

A Web service endpoint is an entity where Web service messages can be targeted. Endpoint references convey the information needed to identify/reference a Web service endpoint. Endpoint references are suitable for conveying the information needed to access a Web service endpoint, but are also used to provide addresses for individual messages sent to and from Web services.

2. message information headers

This defines a family of message information headers that allows uniform addressing of messages independent of underlying transport. These message information headers convey end-to-end message characteristics including addressing for source and destination endpoints as well as message identity.

WS-Reliable messaging uses WS-addressing headers to specify endpoint addresses and convey message related information.

2.1.3 WS-Reliable messaging

Reliable message delivery is a common concept in message oriented communication.

WS-ReliableMessaging specification[8] (WS-RM) describes a protocol that allows messages to be delivered reliably between distributed applications in the presence of network failures. The protocol is described in this specification in a transport-independent manner allowing it to be implemented using different network technologies. To support inter operable Web services, a SOAP binding is defined within this specification.

The protocol defined in this specification depends upon other Web services specifications for the identification of service endpoint addresses and policies. This protocol does not talk about the delivery guarantees and persistence. However WS-RM implementations can provide persistence and delivery guarantees using the available protocol constructs.

WS-Reliable messaging is based on a reliable message model which is given below.

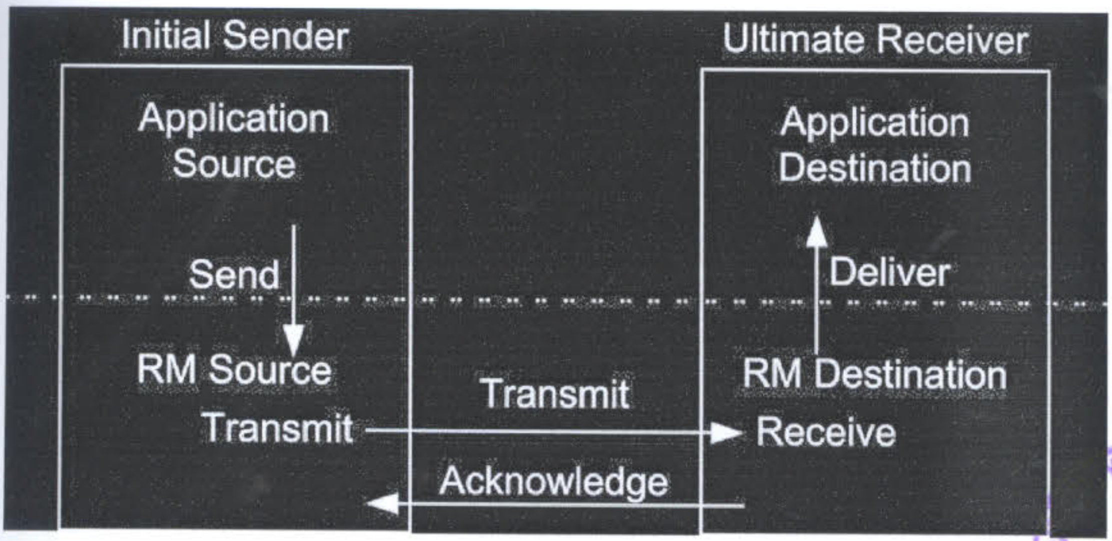


Figure 2-1 Reliable Messaging Model

Following diagram shows the entities and events in a simple reliable message exchange. First, the Application Source sends a message for reliable delivery. The Reliable Messaging (RM) Source accepts the message and Transmits it one or more times. After receiving the message, the RM Destination acknowledges it. Finally, the RM Destination delivers the message to the Application Destination.

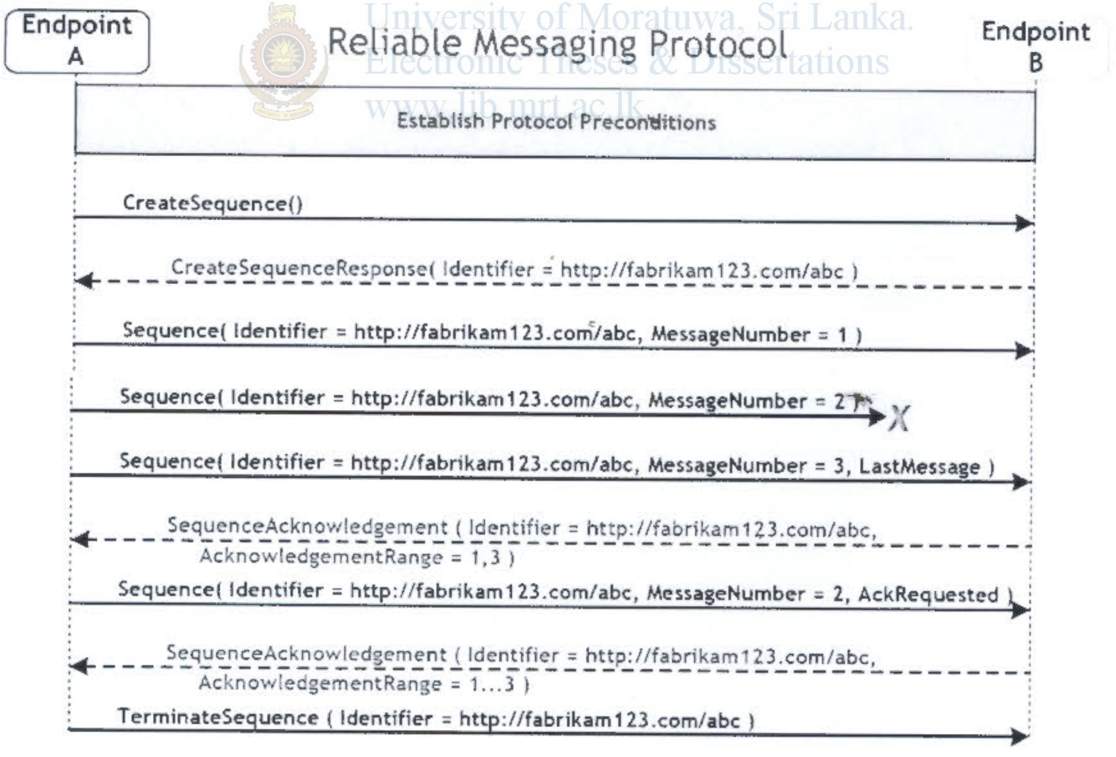

Endpoint A
Reliable Messaging Protocol
Endpoint B
Establish Protocol Preconditions
CreateSequence()
CreateSequenceResponse(Identifier = http://fabrikam123.com/abc)
Sequence(Identifier = http://fabrikam123.com/abc, MessageNumber = 1)
Sequence(Identifier = http://fabrikam123.com/abc, MessageNumber = 2)
X
Sequence(Identifier = http://fabrikam123.com/abc, MessageNumber = 3, LastMessage)
SequenceAcknowledgement (Identifier = http://fabrikam123.com/abc, AcknowledgementRange = 1,3)
Sequence(Identifier = http://fabrikam123.com/abc, MessageNumber = 2, AckRequested)
SequenceAcknowledgement (Identifier = http://fabrikam123.com/abc, AcknowledgementRange = 1...3)
TerminateSequence (Identifier = http://fabrikam123.com/abc)

Figure 2-2 Reliable Messaging Protocol

Following steps illustrates a typical set of messages passed in one RM sequence and how it provides fault tolerance. It uses a acknowledgment based retransmission similar to TCP.

1. The protocol preconditions are established. These include policy exchange, endpoint resolution, establishing trust.
2. The RM Source requests creation of a new Sequence.
3. The RM Destination creates a Sequence by returning a globally unique identifier.
4. The RM Source begins sending messages beginning with MessageNumber 1. In the figure the RM Source sends 3 messages.
5. Since the 3rd message is the last in this exchange, the RM Source includes a <LastMessage> token.
6. The 2nd message is lost in transit.
7. The RM Destination acknowledges receipt of message numbers 1 and 3 in response to the RM Source's <LastMessage> token.
8. The RM Source retransmits the 2nd message. This is a new message on the underlying transport, but since it has the same sequence identifier and message number so the RM Destination can recognize it as equivalent to the earlier message, in case both are received.
9. The RM Source includes an <AckRequested> element so the RM Destination will expedite an acknowledgment.
10. The RM Destination receives the second transmission of the message with MessageNumber 2 and acknowledges receipt of message numbers 1, 2, and 3 which carried the <LastMessage> token.
11. The RM Source receives this acknowledgment and sends a TerminateSequence message to the RM Destination indicating that the sequence is completed and reclaims any resources associated with the Sequence.
12. The RM Destination receives the TerminateSequence message indicating that the RM Source will not be sending any more messages, and reclaims any resources associated with the Sequence.

2.2 Transactions and messaging standards

2.2.1 2PC

Two phase commit protocol[7] is a protocol to support transactions in a distributed environment. In a distributed environment there are multiple participants. These multiple participants update multiple data sources. Two phase commit protocol ensure either these participants commit or abort atomically.

Two phase commit protocol is executed by a process called the coordinator process and other participant processes. As the name suggests two phase commit protocol has two phases called prepare phase and commit phase. Both of these participants' life cycles has been defined by the state transfer diagrams.

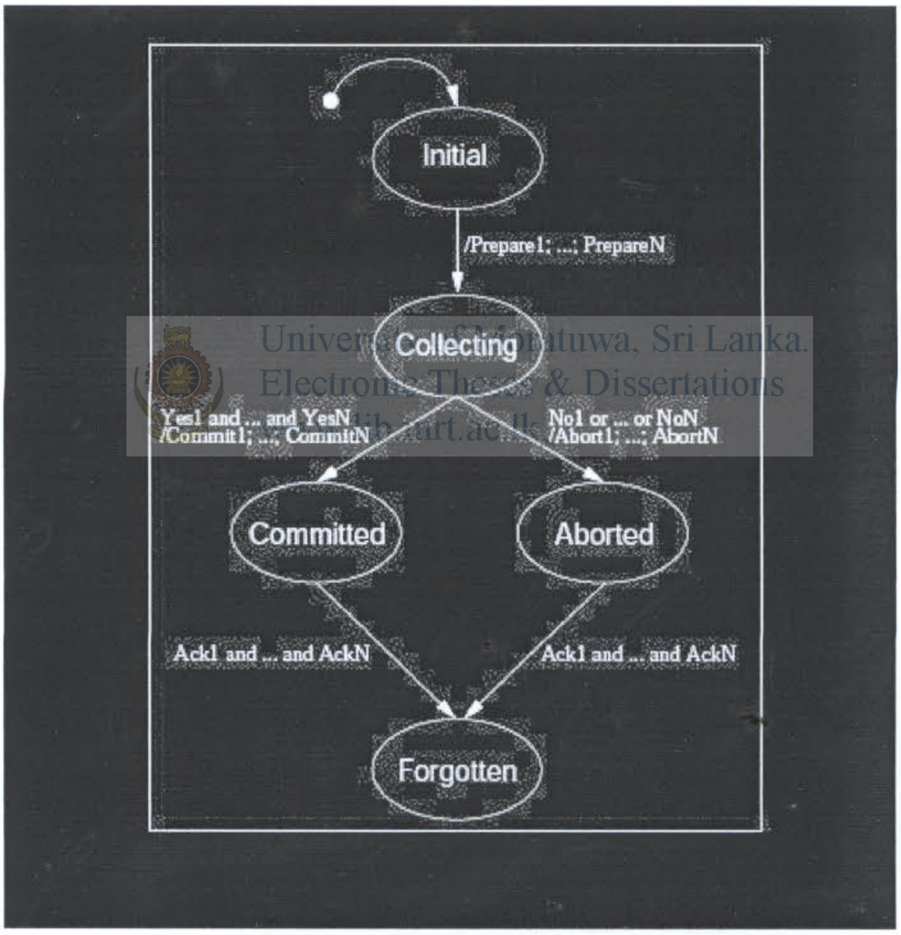


Figure 2-3 Coordinator States

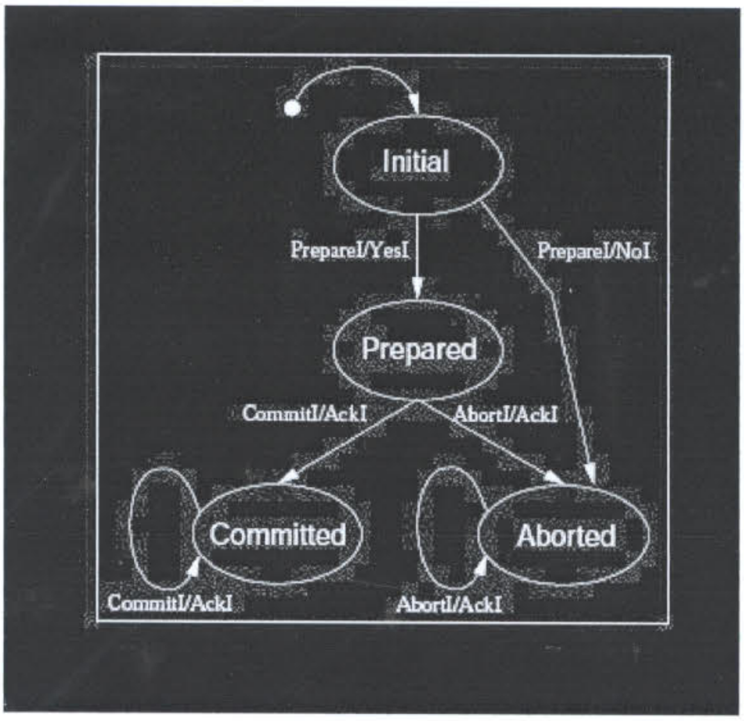


Figure 2-4 Participant States

Before the commit process starts, both coordinator and participants processes are at the initial state. Commit process starts when the initiator sends the commit message to the coordinator. Getting the commit message coordinator sends the prepare message to all the participants and moves to the prepared state and waits until all the responses come. When a participant receives a prepare message from the coordinator it sends the response as 'yes' and moves to prepared state if it is prepared to commit or sends the response as 'no' and moves to aborted state if it is not prepared to commit. Here if a participant sends a 'yes' response it can't later say it is not prepared to commit. Once all the participants sends their responses coordinator can decide either to commit the transaction or abort it. If there is at least one 'no' response coordinator have to decide to abort the transactions. After that coordinator tells its participants either to abort or commit and then moves to either commit or abort state. Once the participants gets the global commit or abort message from the coordinator it moves to the corresponding state and sends the acknowledgment back to the coordinator.

2.2.2 X/Open distributed transaction standards

X/Open is a independent, worldwide, open systems organization which supports implementation of open systems. In the context of the distributed transactions, X/Open has standardize the interface between the Transaction Manager and the Resource Manager in order to make them as open systems[11].

X/Open distributed transaction processing (DTP) model assumes three software components.

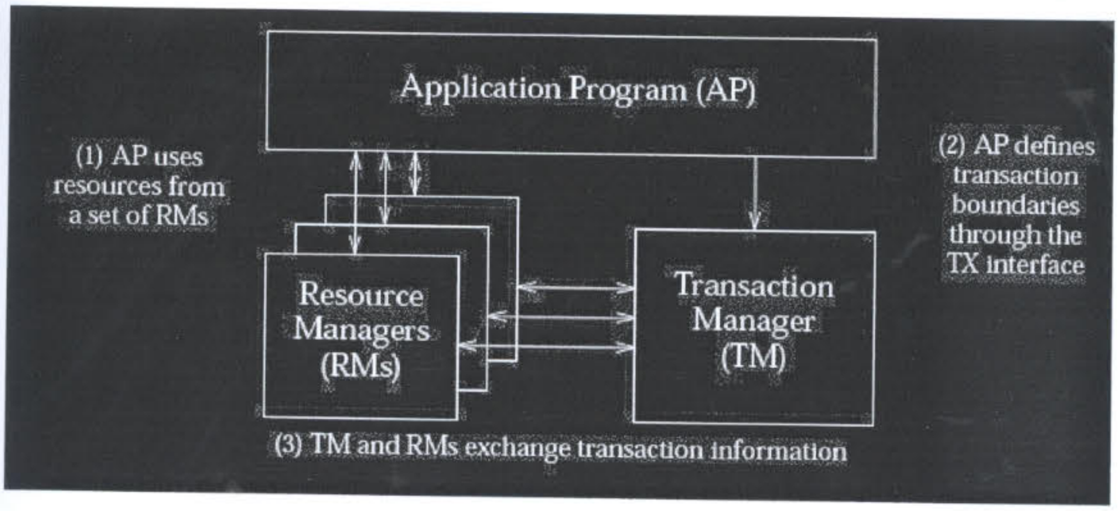

Application Program (AP)
(1) AP uses resources from a set of RMs
(2) AP defines transaction boundaries through the TX interface
Resource Managers (RMs)
Transaction Manager (TM)
(3) TM and RMs exchange transaction information

Figure 2-5 X/Open Distributed Transaction Standards

Application program specify the transaction boundaries and specifies the actions that constitute the transaction. Resource managers provides the resources which application program updates during a transaction. Transaction manager is the main component which assigns identifiers to transactions, monitor their progress and do the transaction completion or failure recovery.

Out of these interactions X/Open specification introduces a standard interface to communicate between the Transaction manager and the Resource managers. These interfaces are specified in C programming language.

2.2.3 JTA

Java transaction API specification[12] provides a set of java interfaces to support distributed transactions. It specifies the local Java interfaces between a transaction manager and the parties involved in a distributed transaction systêm. Following diagram shows the interfaces it defines and the relevant areas of those specifications.

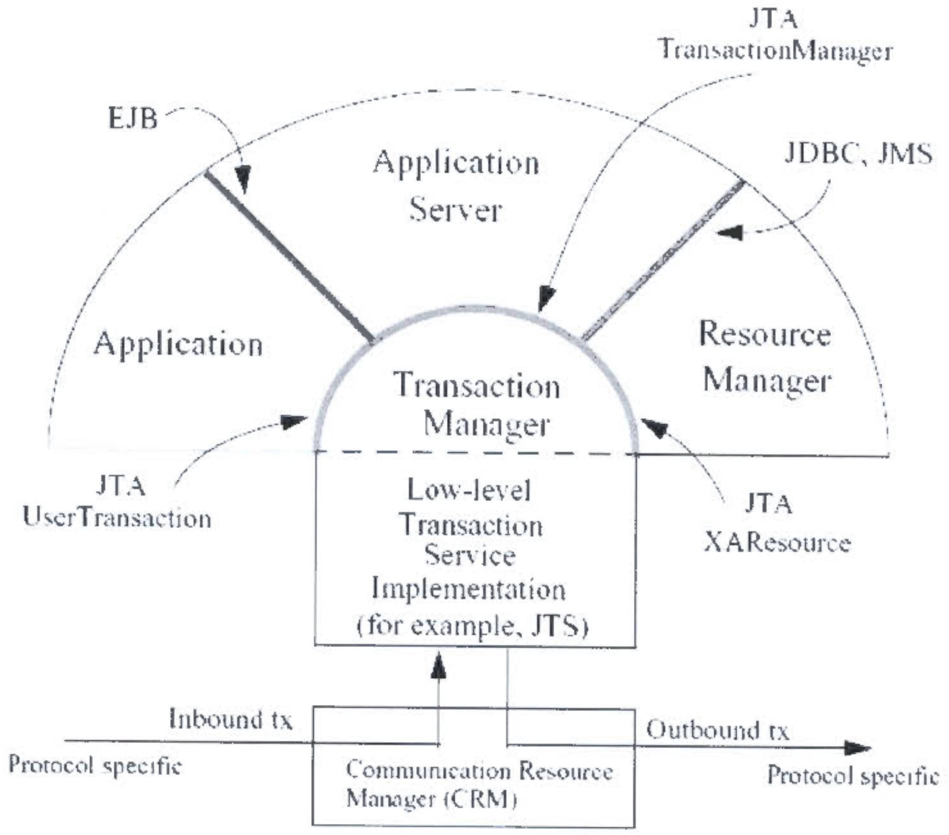


Figure 2-6 JTA Overview

UserTransaction interface provides the application the ability to control the transaction boundaries programmatically. The application can obtain user transaction and use begin and commit method to demarcate the transactions.

Transaction manager interface allows application server to control transaction boundaries. Transaction Manager allows users to begin and commit transactions associated with a thread.

Transaction interface allows operations to be performed on the transaction associated with target object. This interface can be used to

1. Enlist the transactional resources in use by the application
2. Register for transaction synchronization callbacks
3. Commit or rollback the transaction

XAResource Interface provides a java mapping of the industry standard XA interface based on the X/Open Specification. This interface defines the contracts between the Resource Manager and the Transaction Manager in a distributed transaction processing (DTP) environment.

JTA specification defines five players which are involved in a distributed transaction services. Each of these players contribute to the distributed transaction processing system by implementing different sets of transaction API and functionalities.

1. A transaction Manger provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.
2. A application server provides the infrastructure required to support the application run time environment which includes transaction state management.
3. A resource manager provides the application access to resources.
4. User application which uses the transaction provided by the application server.
5. A communication resource manager supports transaction context propagation and access to the transaction service for incoming and outgoing requests.

2.3 WS-Transactions

WS-Transactions defined in three specifications. WS-Coordination defines a common framework to coordinate web services activities among different web services using different types of coordinating protocols.

2.3.1 WS-Coordination

WS-Coordination[4] describes an extensible framework for providing protocols that coordinate the actions of distributed applications. Such coordination protocols are used to support a number of applications, including those that need to reach consistent agreement on the outcome of distributed activities.

The framework defined in this specification enables an application service to create a context needed to propagate an activity to other services and to register for coordination protocols. The framework enables usage of existing proprietary transaction processing systems while providing an inter operable mechanism to communicate.

The following diagram shows typical usage scenario of the WS-Coordination specification to coordinate the activities among different web services.

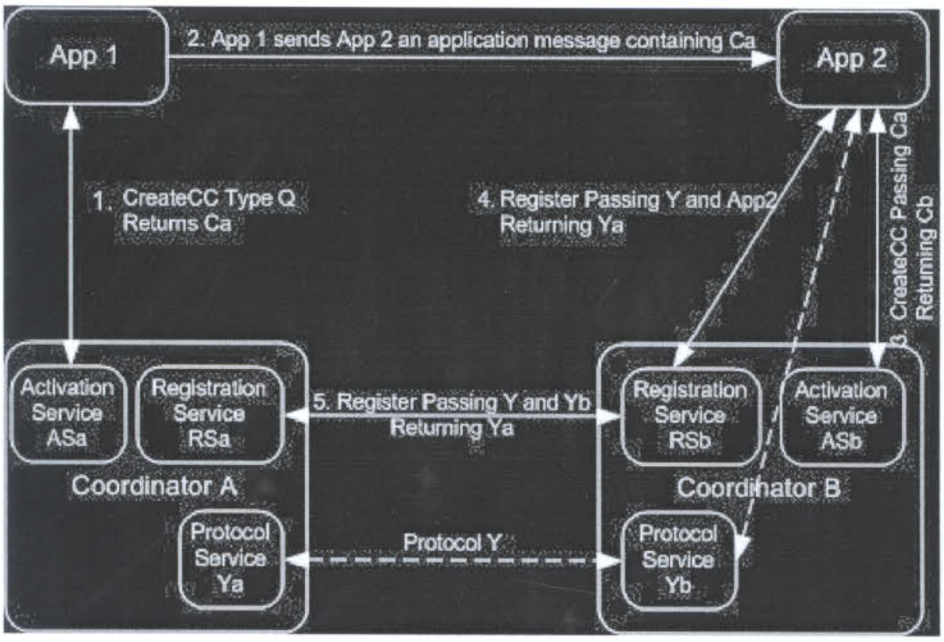

App 1
2. App 1 sends App 2 an application message containing Ca
App 2
1. CreateCC Type Q Returns Ca
4. Register Passing Y and App2 Returning Ya
3. CreateCC Passing Ca Returning Cb
Activation Service ASa
Registration Service RSa
5. Register Passing Y and Yb Returning Ya
Registration Service RSb
Activation Service ASb
Coordinator A
Coordinator B
Protocol Service Ya
Protocol Y
Protocol Service Yb

Figure 2-7 WS Coordination Framework

1. App1 sends a CreateCoordinationContext for coordination type Q, getting back a Context Ca that contains the activity identifier A1, the coordination type Q and an Endpoint Reference to CoordinatorA's Registration service Rsa.
2. App1 then sends an application message to App2 containing the Context Ca.
3. App2 prefers CoordinatorB, so it uses CreateCoordinationContext with Ca as an input to interpose CoordinatorB. CoordinatorB creates its own CoordinationContext Cb that contains the same activity identifier and coordination type as Ca but with its own Registration service RSb.
4. App2 determines the coordination protocols supported by the coordination type Q and then Registers for a coordination protocol Y at CoordinatorB, exchanging Endpoint References for App2 and the protocol service Yb. This forms a logical connection between these Endpoint References that the protocol Y can use.
5. This registration causes CoordinatorB to forward the registration onto CoordinatorA's Registration service RSa, exchanging Endpoint References for Yb and the protocol service Ya. This forms a logical connection between these Endpoint References that the protocol Y can use.

2.3.2 WS-Atomic transactions

WS-Atomic transactions specification[5] defines an atomic transaction coordination type that can be used with the WS-Coordination specification. This specification describes such coordination type protocols which can be used with the short lived atomic transactions.

Completion

This protocol is used to communicate between the initiator and the coordinator. Initiator starts the commitment processing by sending a commit message. After that coordinator starts the volatile 2PC and proceed to durable 2PC. Then the final result is send to the initiator.

Two phase commit protocol

Two phase commit protocol is used to perform the atomic transaction among the participators. This protocol ensures all the participators comes to a final decision. There are two variations of this protocol.

1. Volatile two phase commit

Used with the participators who use the volatile resources such as memory cache.

2. Durable two phase commit

Use with the participators use the durable resources such as databases.

2.3.3 WS-BussinessActivity

Similar to WS-Atomic transactions specification this specification also defines coordination types and protocols to be used with WS-Coordination specification. These coordination types typically has to be used with the long running transactions. There are two coordination types and protocols has defined in this specification.

Coordination types

There are two coordination types have defined with this specification called atomic outcome and mixed outcome. In the atomic outcome coordination type all the participators either end up with end state or compensated state while in the mixed outcome mode participators and be end up within any state .

Coordination protocols

There are two types of coordination protocols defined with this specification called BusinessAgreementWithParticipantCompletion and BusinessAgreementWithCoordinatorCompletion. The former protocol initiation starts by the participant while for the latter it is started by the coordinator.

2.4 Queued Transaction processing

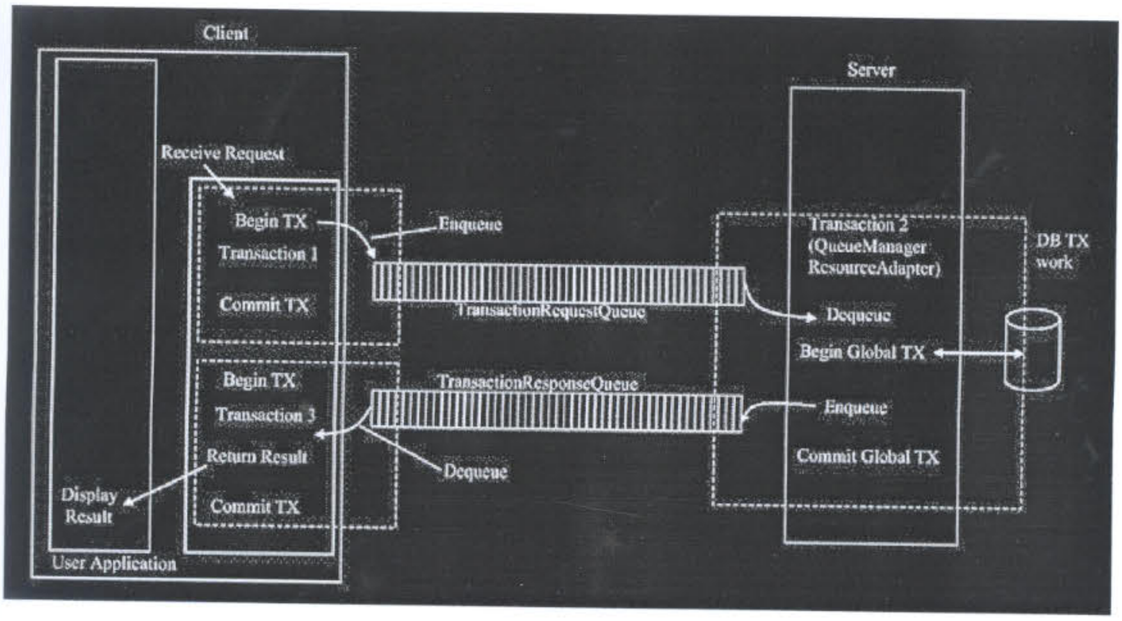


Figure 2-8 Queued Transaction Processing

Queued transaction processing is used to process transactions in a distributed environment asynchronously. This happens within three transaction boundaries. Firstly user application creates the request message and enqueues the request message to request queue within a transaction. After that server dequeues the message, process it and enqueues the response to response queue within another transaction. Finally user application dequeues the message from the response queue.

2.5 Different types of reliable web services

2.5.1 Using Message Oriented Middleware for Reliable Web Services Messaging.

Web services are applications that are described, published and accessed over the web using open XML standards. Different Message Oriented Middleware can be used with web services. Reliable communication is one of the most important aspects of any application. There are five ways that an web service can use MOM.

1. Messaging Middleware Reliability

Messaging middleware is specialized software that accepts messages from sending processes and delivers them to receiving processes. The two principle styles for MOM is centralized and distributed.

2. Aspects of reliability

The main aspect of the reliability is to tolerate the network failures. MOM can tolerate the network failures by repeatedly sending the message until it is acknowledged by the receivers component. In addition to acknowledged delivery, ordered delivery is another aspect of reliable messaging. Further important aspect of reliability is the integration of a message delivery in a larger processing context. Therefore a MOM should be able to group a message with other messages and other process activities.

2.5.2 Three facets of Reliability

1. Middleware endpoint to endpoint reliability

A message once delivered from an application to the messaging middleware, is guaranteed to be available for consumption by the receiving process.

2. Application to middleware reliability

The middleware's messaging API, supports reliability properties such as message delivery guarantees, message persistence and transactional messaging.

3. Application to application reliability

Sending and receiving applications engage in transactional business processes that rely on application-to-middleware reliability and middleware endpoint-to-endpoint reliability.

2.5.3 Reliable messaging for web services

This describes five different ways in which a web service can use the MOM for a reliable communication.

1. SOAP (with or without a reliability protocol like WS-ReliableMessaging) is used with an unreliable transport (like Http); reliability mechanisms are implemented on the application/SOAP messaging layer.
2. A Reliable transport like HTTPR is used for SOAP messaging
3. A Reliable, proprietary middleware system like IBM Websphere MQ is used for SOAP messaging.
4. A Reliable messaging standard like JMS is used for SOAP messaging. A JMS implementation is required
5. A Reliable proprietary middleware system like IBM Webpshere MQ is directly used independent of SOAP

2.5.4 Assessment

Middleware endpoint to endpoint reliability

The middleware endpoint mediation essentially refers to the idea that messages are stored locally on the sender and receiver sides before and after they are being sent.

1. Option 1 does not provide this reliability since HTTP is not reliable. HTTP does not provide the status of the message on a connection failure. Therefore either SOAP messaging layer or application layer should provide the reliability.
2. SOAP over HTTPR provides the middleware endpoint to endpoint reliability. HTTPR persists the messages at the sender and receiver sides.
3. SOAP over MQ also provides the middleware endpoint to endpoint reliability. The middleware endpoints are message queue managers provided by the messaging middleware product. Unlike in the HTTPR case here the message delivery pattern is asynchronous.
4. SOAP over JMS requires a JMS implementation. Depending on the JMS implementation it provides the reliability.
5. This option also supports the reliability since underline MOM is reliable. Adapters must be used at the each side to send and receive XML messages at each side.

Application to Middleware reliability

Application to middleware reliability refers to the reliability features provided by the middleware's application to endpoint interface. This includes message delivery guarantees, fault tolerant invocation, the ability to atomically group messaging operations with other application actions.

1. When using SOAP over HTTP the reliability mechanisms may be implemented as part of the application. Application can transactionally coordinate with the message store to guarantee the reliability.
2. For options 2 – 4 applications can't communicate transactionally with MOM message store without using the MOM specific APIs.
3. For last option application to middleware reliability relates to the direct use of the underlying middleware's API and its reliability features.

Application to application reliability

Application to application reliability can be achieved in two ways.

1. In direct transaction processing, an agreement protocol is used to directly include one application's transaction processing as part of another application's transaction process. Here both applications interact with the same global transaction.
2. In Queued transaction processing two intermediate data stores can be used for sending and receiving messages. There are three transactions involve in communication between two applications. First transaction commits the message to sending data store. Then the receiving application reads the message from there and commit back to the second storage. Finally original sending application reads the response message from the second storage.

2.6 Transactions and messaging

Messaging can be integrated with the object transactions in different ways. This paper[16] pointed out such for patterns possibly used.

2.6.1 MQ Integrating Transactions

MQ Integrating transactions do the reading messages from the queue, updating the distributed object and writing the response message back to the queue in the same transaction. But this transaction corresponds only a part of the global transaction.

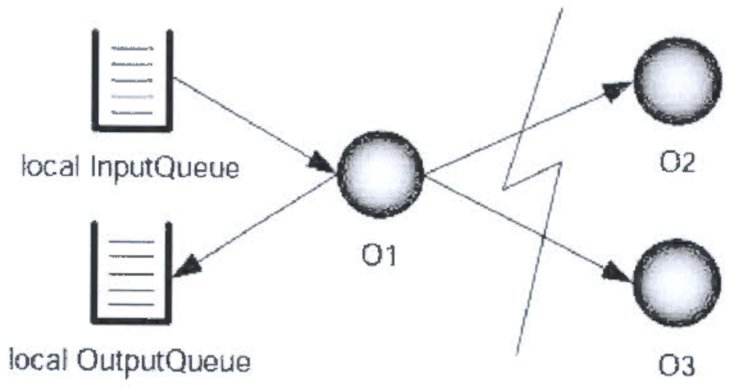


Figure 2-9 MQ Integrating Transactions

2.6.2 Message delivery transactions

Message delivery transactions integrates the message delivery model into distributed transactions. It allows clients to send the messages asynchronously while doing the other distributed object transactions. Message delivery failures can be observed and abort the transaction accordingly. If there are messages already sent then compensation messages can be send.

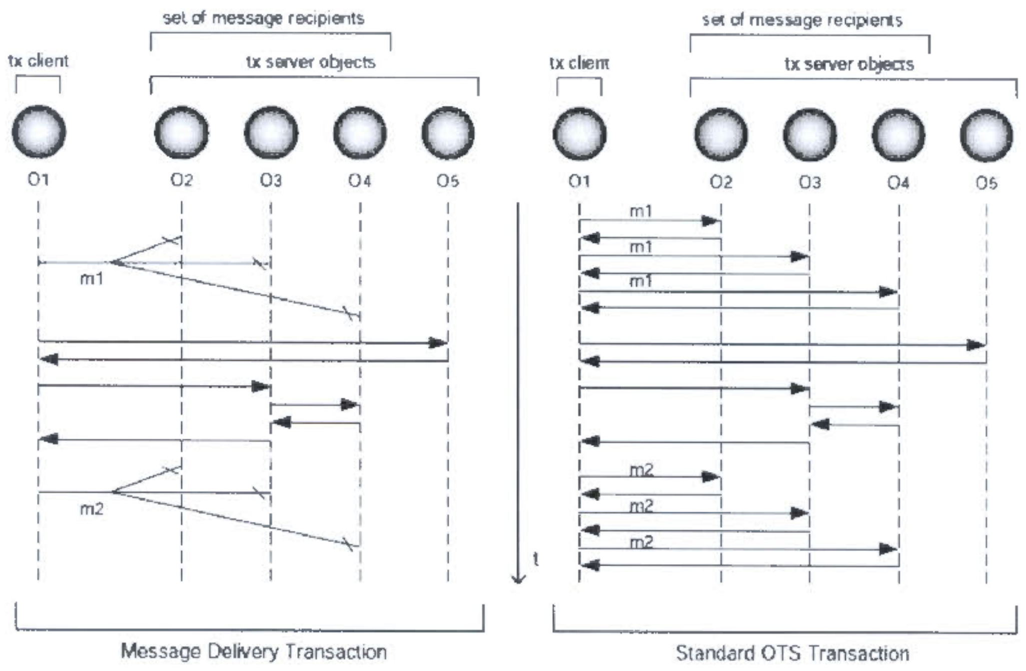


Figure 2-10 Message Delivery Transactions

2.6.3 Message processing transactions

Message processing transactions integrates the message processing model to the distributed object transactions. This enables the asynchronous request processing between transactional distributed objects. The transaction is not committed until the response is received.

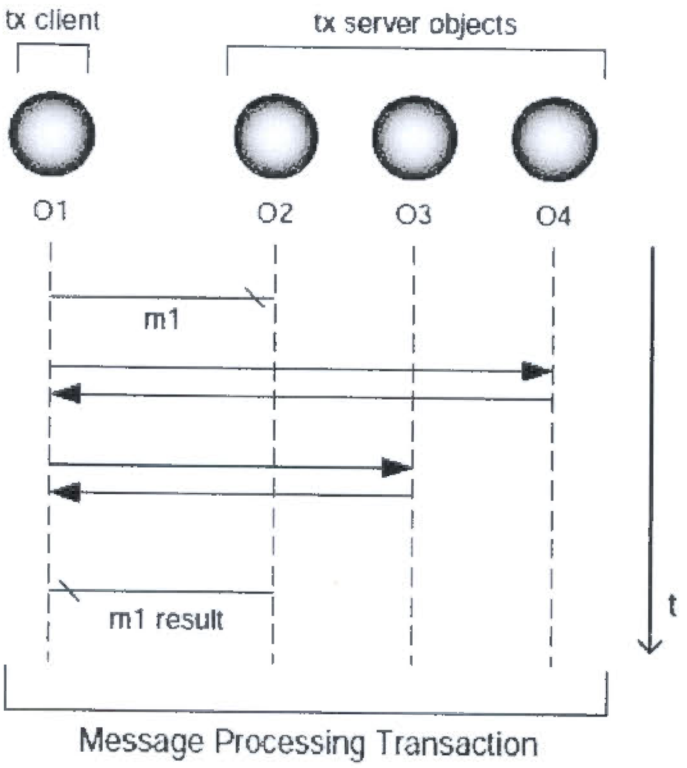


Figure 2-11 Message Processing Transactions

2.6.4 Full messaging transactions

Full messaging transactions refers to the system which has both the message delivery transactions as well as the message processing transactions.

2.7 Middleware mediated transactions

There are two widely used transaction processing systems called. Object oriented transactions and message oriented transactions. Object oriented transactions happens in a synchronous blocking way . Further object oriented transactions uses 2PC protocol to achieve the atomicity of the transactions. In message oriented transactions only enqueuing and dequeuing messages are done transactionally. Therefore message oriented transactions does not preserve the atomicity.

Middleware mediation transactions[15] suggest a way to provide the end to end transactions while keeping the advantages of the message mediation transactions. It provides some end to end checking at the middleware layer.

2.7.1 D-sphere

D-sphere[17] is one of the implementations of the middleware mediation transactions. D-sphere provides the end to end reliability by providing an middleware to the user which manages the end to end transactions. Following figures show how it works with and without D-sphere.

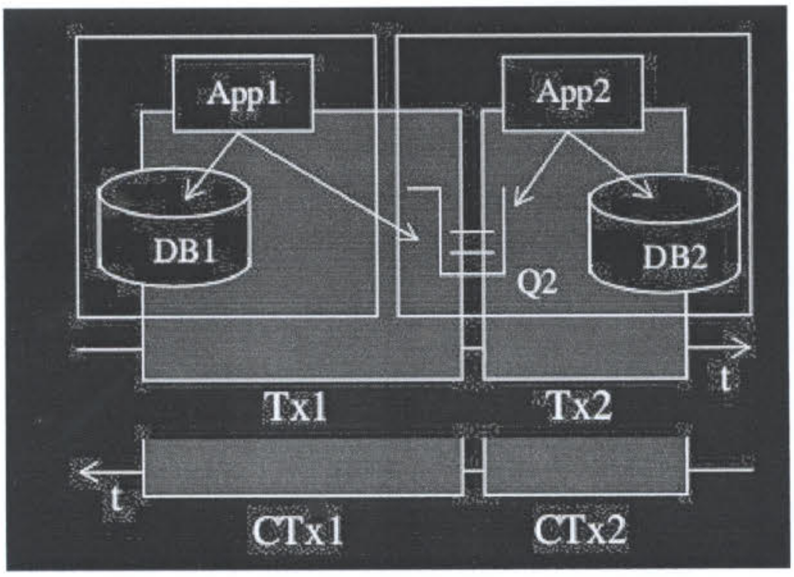


Figure 2-12 Application Without D-sphere

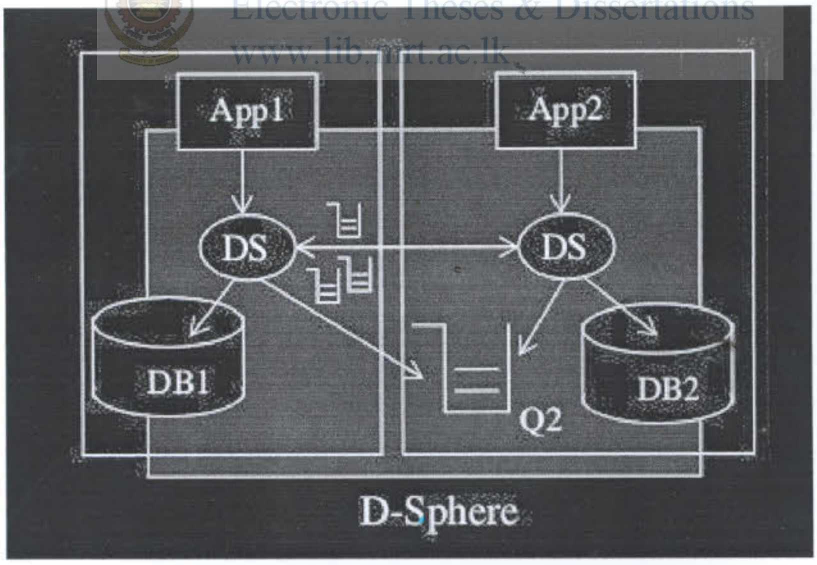


Figure 2-13 Application With D-sphere

D-sphere architecture supports above requirements by providing a middleware layer to users which manage transactions internally.

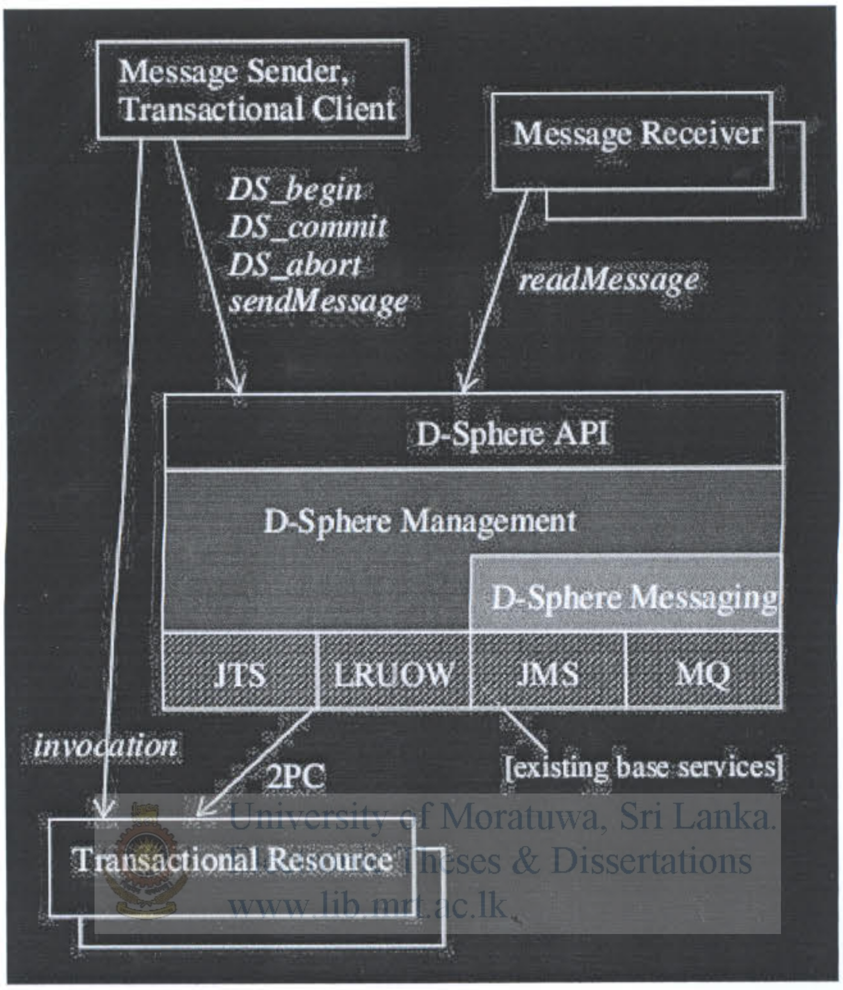


Figure 2-14 D-sphere Architecture

Chapter 3

Methodology

3.1 Previous Solution

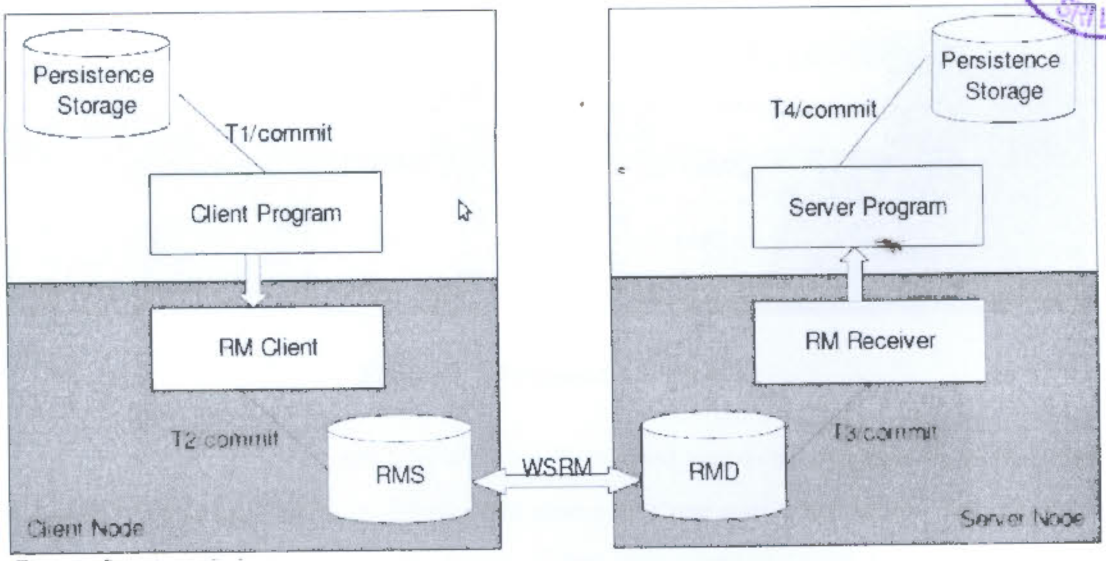


Figure 3-1 Previous Solution

Fig 3-1 shows the probable way of solving the above problem with the original WSO2 Mercury implementation. First the client program has to read its' persistence storage within a transaction, build the message in memory and commit the transaction. Then it gives the message to RMClient which again has to start a transaction and commit the message to RMS. Once the message stored at RMS it reliably transferred to the RMD by the WS-RM protocol. At the server side RMReceiver receives the message within a transaction from the RMD, build the message in memory and commit the transaction. Finally server program commits the message to server persistence storage within a transaction.

This model reliably operates with the presence of network failures since WS-RM protocol can handle it using retransmissions and acknowledgements. But if the client node fails after first transaction being committed to the client storage and before RMClient commit it to the RMS then the message can be lost. Same failure can occur at the server side as well. On the other hand if the first transaction commits after the second transaction there can be duplicated messages.

3.2 Proposed Solution

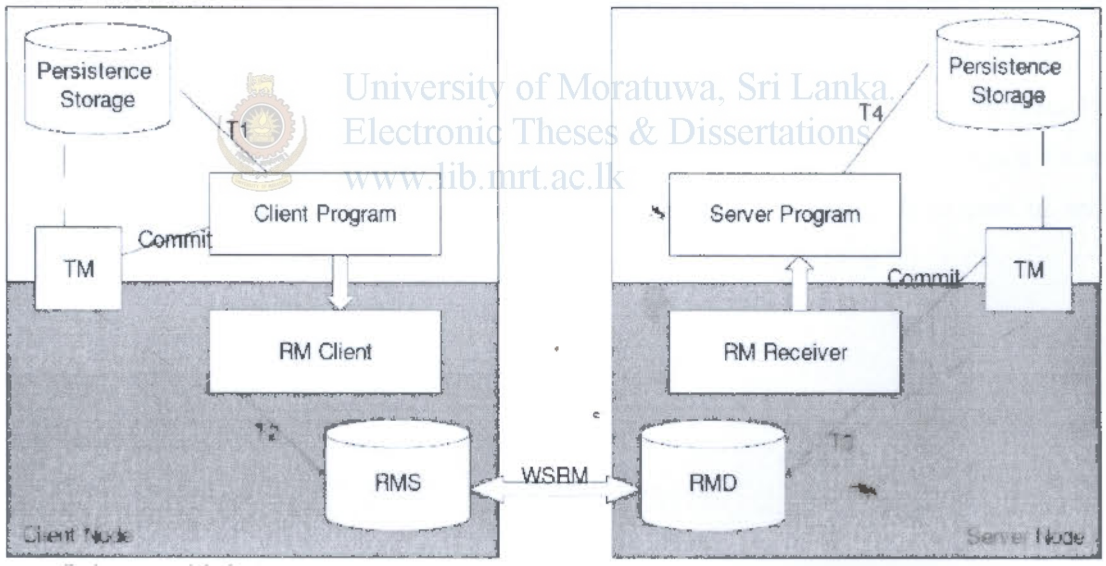


Figure 3-2 Proposed Solution

Fig 3-2 shows the proposed solution with the distributed transaction support to address the node failure scenarios. Unlike in the previous case now client uses a distributed transaction to update both client storage and RMS storage. Client only commits to the transaction manger and if client node fails when this commit happens, the recovery process of the 2PC ensures the atomicity of the transaction. Similar process happens at the RMReceiver as well. At the server side server program transaction has to participate the distributed transaction started by the RMReceiver and again node failure handle by the recovery process of the 2PC protocol.

3.3 Alternative Solutions

3.3.1 Integrate WS-RM protocol with the client storage and server storage by taking them as RMS and RMD

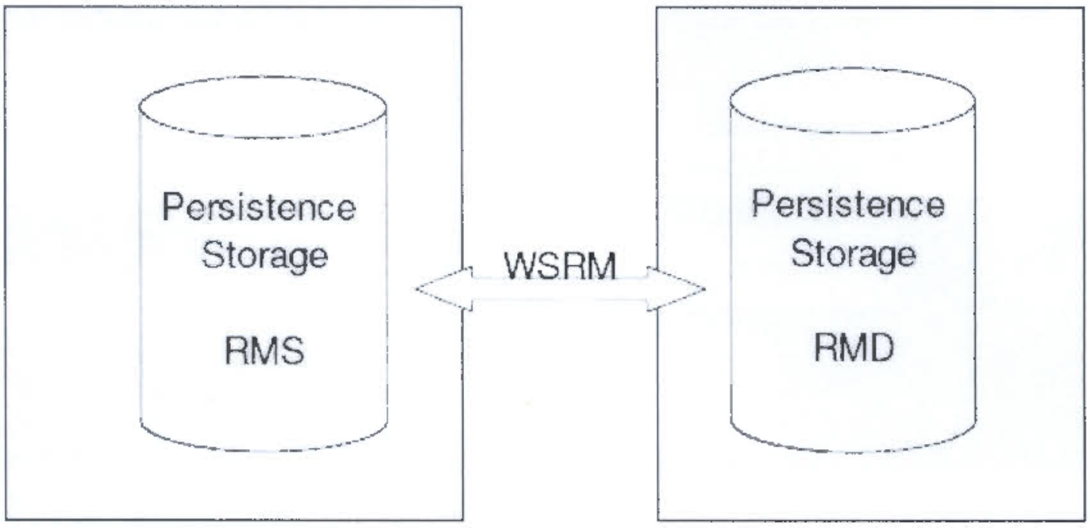


Figure 3-3 Integrate Persistence Storage with RM Storages

Fig 3-3 shows a possible solution for this problem in a specific way to a given problem. For this solution distributed transactions are not required since RM protocol tightly integrated to the client and server storages. But the advantages of this type of approach is less since it does not try to solve the problem in a generalized way. The focus of this project is to come up with a middleware which provides the WS-RM functionality to any application.

3.3.2 Use the same transaction to update both application level storages and WS-RM storages

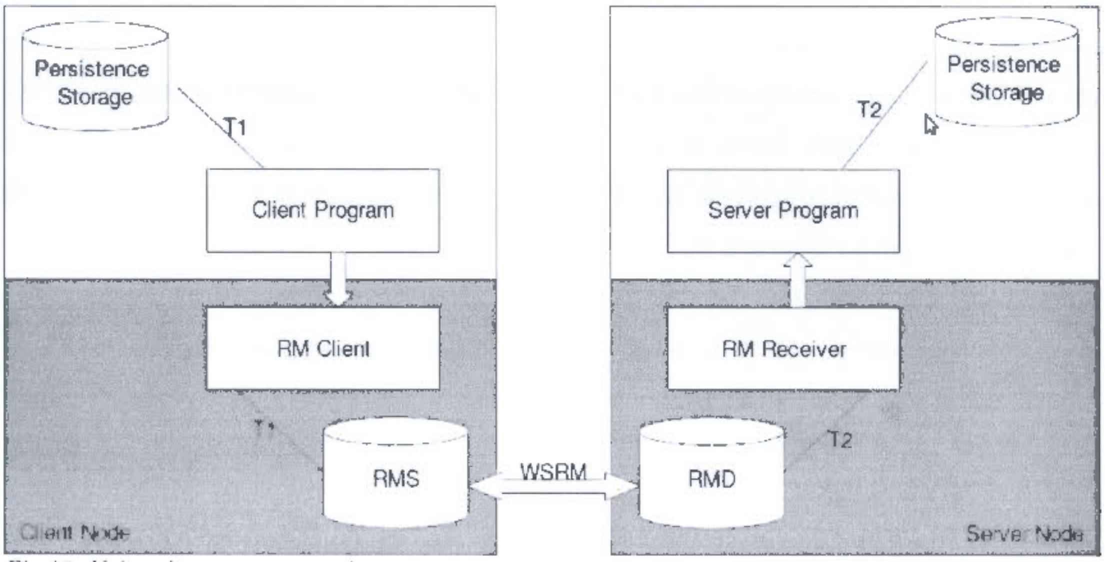


Figure 3-4 Using the same Transaction

Fig 3-4 shows a special case of using same transaction to update both application level storages and the WS-RM storages. In order to use this scenario WS-RM storages should be there with the same application storages. Therefore this may not be useful when integrating message receive functionality with different storages and different application servers. Although this functionality can be provided with the proposed storage API based Mercury implementation, this project only focus on the distributed transaction based solution.

3.4 Solution Architecture

3.4.1 State machine model

WSO2 Mercury is based on a state machine model where the state is kept in objects called RMSSequence, RMDSequence and InvokerBuffer. This state machine model is based on the fact that various external events change the state of each object. Further a set of workers namely RMSSequenceWorker, RMDSequenceWorker and InvokerBufferWorker performs set of actions based on the state of the object. It does not assume any order of the events. If a WS-RM message get lost while transmitting through the network, only the event which would have occurred get lost while system state remains same. Therefore system operates in the previous state which causes the retransmission of lost message hence achieving reliability.

RMSSequence

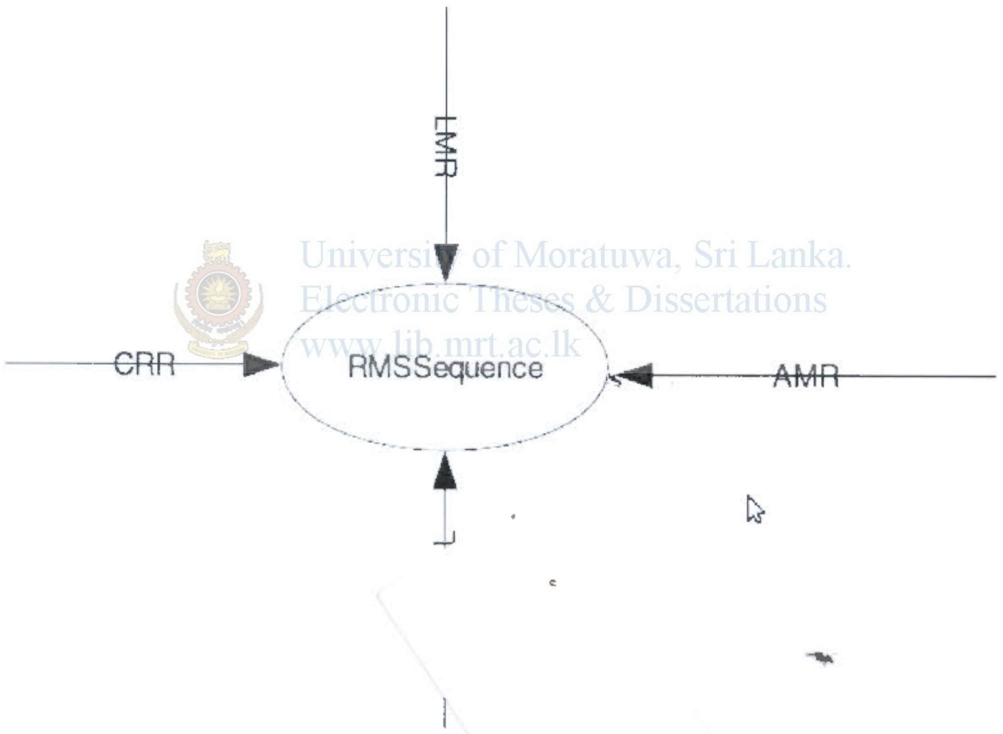


Figure 3-5 RMSSeq

Fig 3-5 shows the possible events that would change tate. These events namely create sequence response receive (CRR), last m 'R), application message receive (AMR) and receive acknowledgement for a her cause by the application client or network message reception. RMSSe 's on four factors called sequence started (SS), message in the buffer (MIB), (LMR) and terminate message send (TMS). These four factors create possib only seven states are valid as shown in the Fig 3-6.

[illegible]

SS	MIB	LMR	TMS
~~0~~	~~0~~	~~0~~	~~0~~
~~0~~	~~0~~	~~0~~	~~1~~
~~0~~	~~0~~	~~1~~	~~0~~
~~0~~	~~0~~	~~1~~	~~1~~
0	1	0	0
~~0~~	~~1~~	~~0~~	~~1~~
0	1	1	0
~~0~~	~~1~~	~~1~~	~~1~~
1	0	0	0
~~1~~	~~0~~	~~0~~	~~1~~
1	0	1	0
1	0	1	1
1	1	0	0
~~1~~	[illegible]	[illegible]	~~1~~
1	[illegible]	[illegible]	0
~~1~~	~~1~~	[illegible]	~~1~~

Figure 3-6 RM[illegible]

[illegible]

Fig 3-7 shows the complete state transition diagram [illegible] and events which change those states.

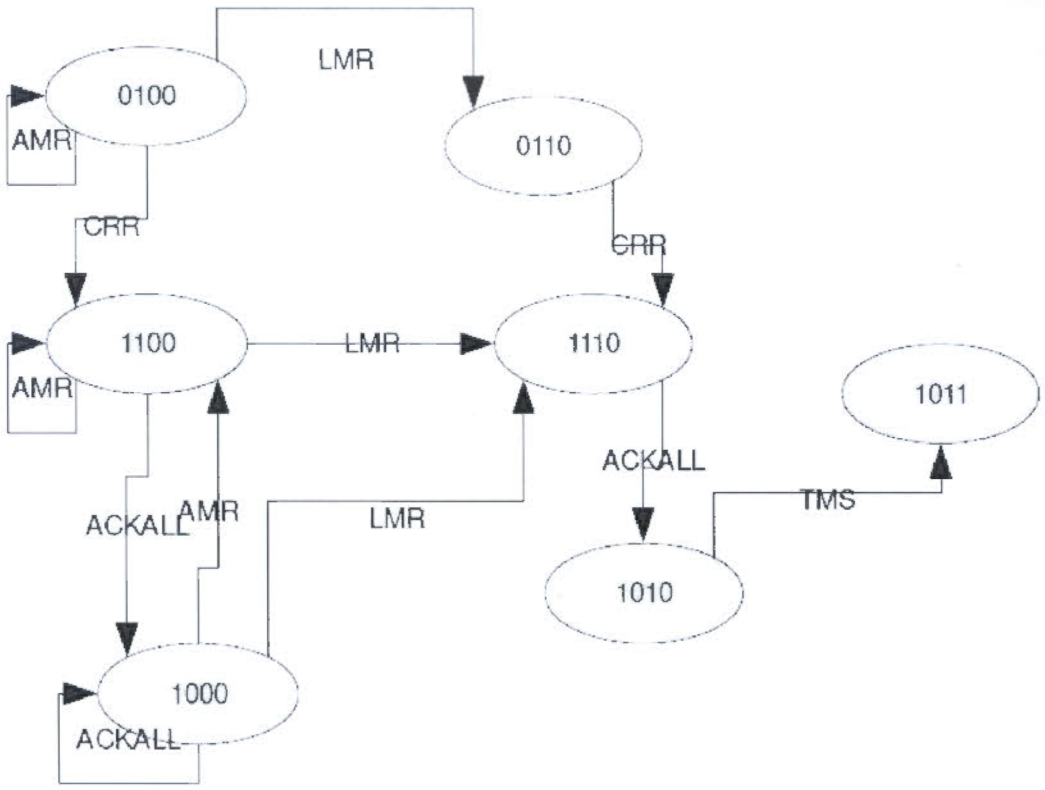


Figure 3-7 RMSSequence State Machine

[illegible]

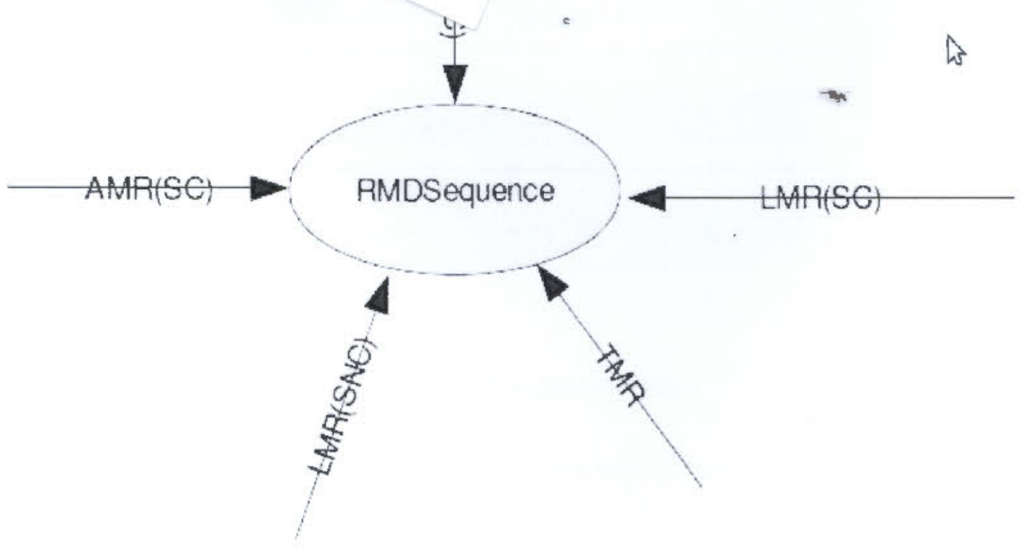
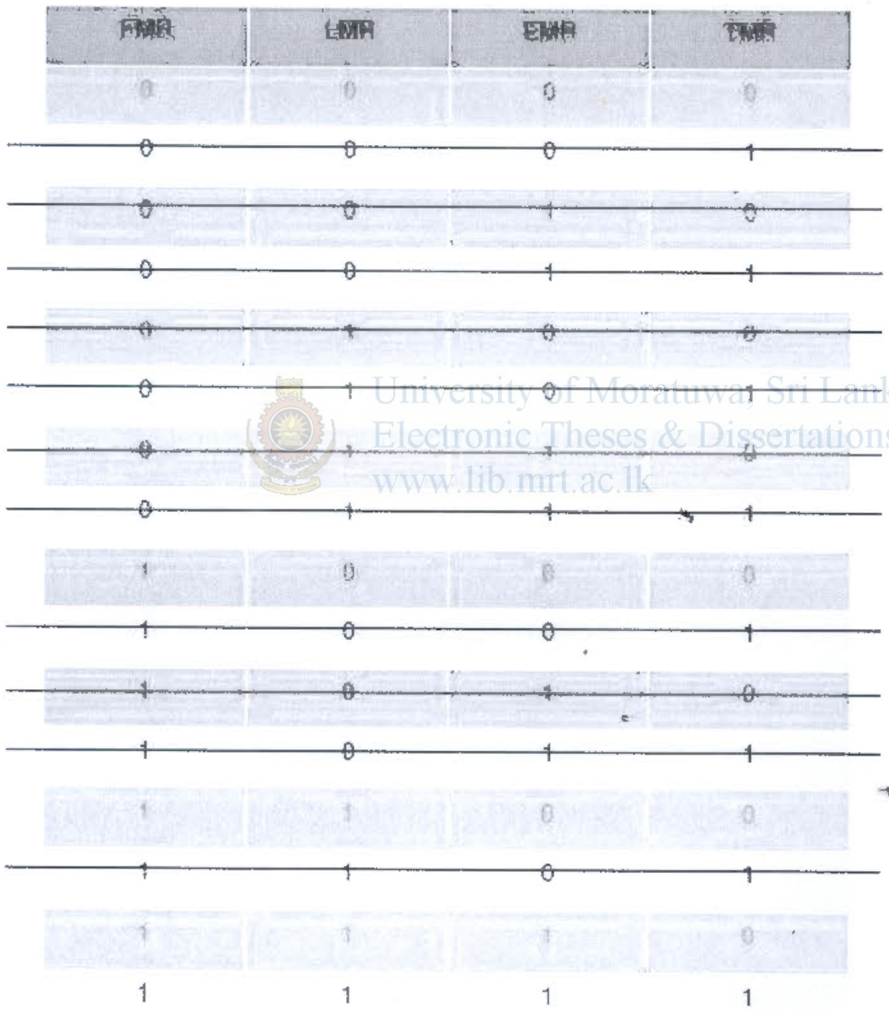


Figure 3-8 RMDSequence Events

Fig 3-8 shows the possible events that would change the RMDSequence state. These events namely application message receive completing the sequence (AMR(SC)), application message receive without completing the sequence (AMR(SNC)), last message receive completing the sequence (LMR(SC)), last message receive without completing the sequence (LMR(SNC)) and terminate message receive (TMR) would cause by the message receive from the network. RMDSequence state depends on four factors called first message receive (FMR), last message receive (LMR), every message has received (EMR) and terminate message receive (TMR). These four factors forms possible sixteen states but only five states are valid as shown in the Fig 3-9.



FMR	LMR	EMR	TMR
0	0	0	0
~~0~~	~~0~~	~~0~~	~~1~~
~~0~~	~~0~~	~~1~~	~~0~~
~~0~~	~~0~~	~~1~~	~~1~~
~~0~~	~~1~~	~~0~~	~~0~~
~~0~~	~~1~~	~~0~~	~~1~~
~~0~~	~~1~~	~~1~~	~~0~~
~~0~~	~~1~~	~~1~~	~~1~~
1	0	0	0
~~1~~	~~0~~	~~0~~	~~1~~
~~1~~	~~0~~	~~1~~	~~0~~
~~1~~	~~0~~	~~1~~	~~1~~
1	1	0	0
~~1~~	~~1~~	~~0~~	~~1~~
1	1	1	0
1	1	1	1

Figure 3-9 RMDSequence States

Fig 3-10 shows the complete state transition diagram with the set of valid states and events which change those states.

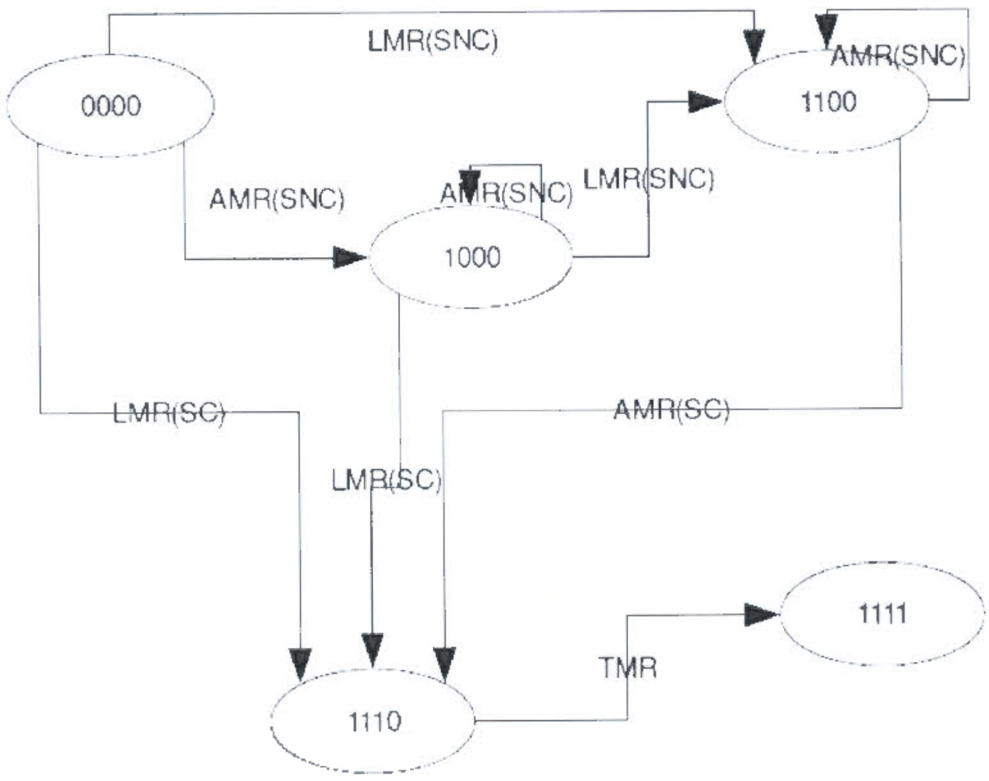


Figure 3-10 RMDSequence State Machine

InvokerBuffer

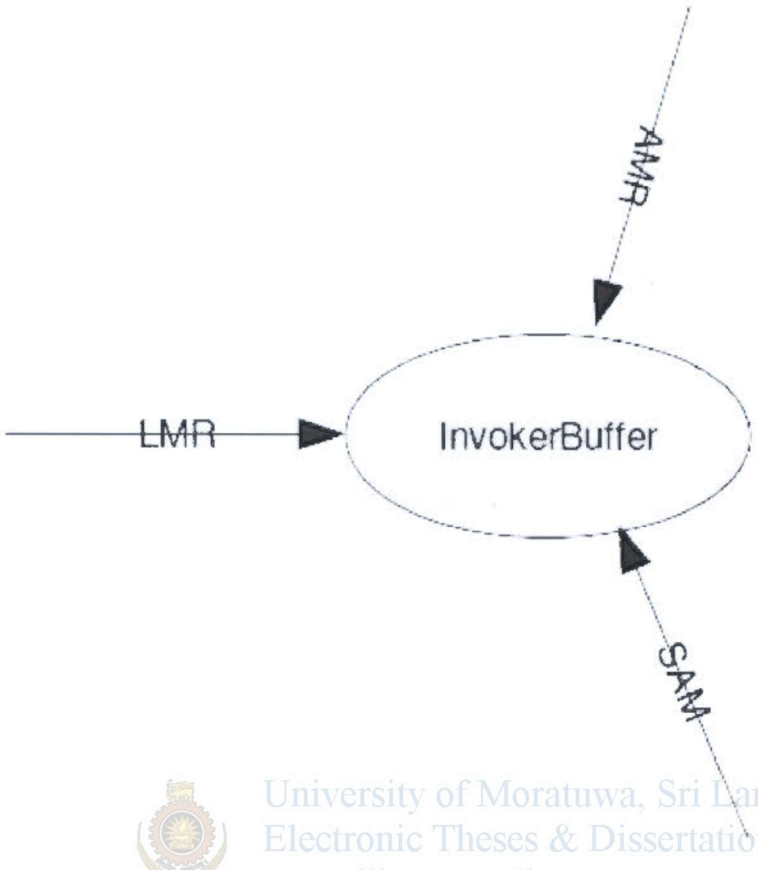
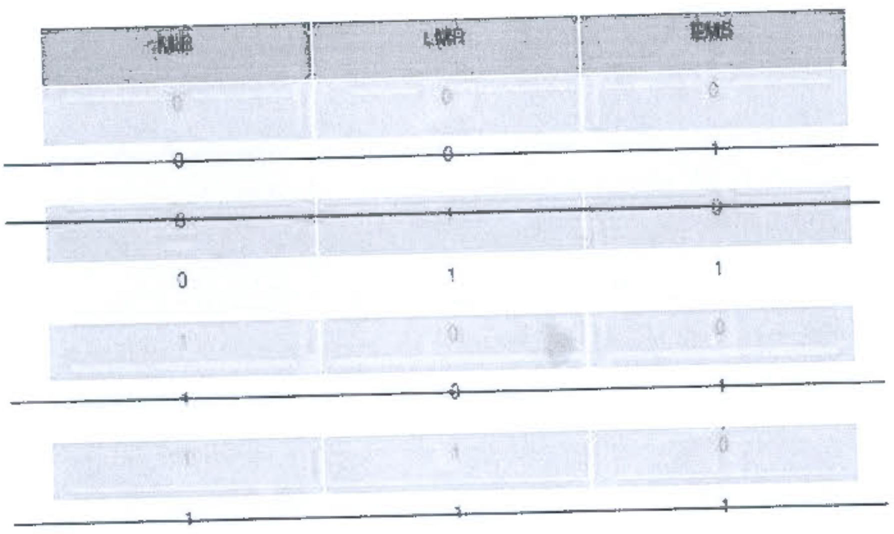


Figure 3-11 InvokerBuffer Events

Fig 3-11 shows the possible events that would change the InvokerBuffer state. These events namely last message receive (LMR), application message receive (AMR) and send all available messages to application (SAM) can cause by the messages receive through the network or the invoker which sends the message to application layer. InvokerBuffer state depends on three factors namely messages in the buffer (MIB), last message received (LMR) and every message send (EMS). These three factors forms possible eight states but only four states are valid as shown in the Fig 3-12.



AMR	LMR	EMS
0	0	0
~~0~~	~~0~~	~~1~~
~~0~~	~~1~~	~~0~~
0	1	1
1	0	0
~~1~~	~~0~~	~~1~~
1	1	0
~~1~~	~~1~~	~~1~~

Figure 3-12 InvokerBuffer States

Fig 3-13 shows the complete state transition diagram with the set of possible states and events which change those states.

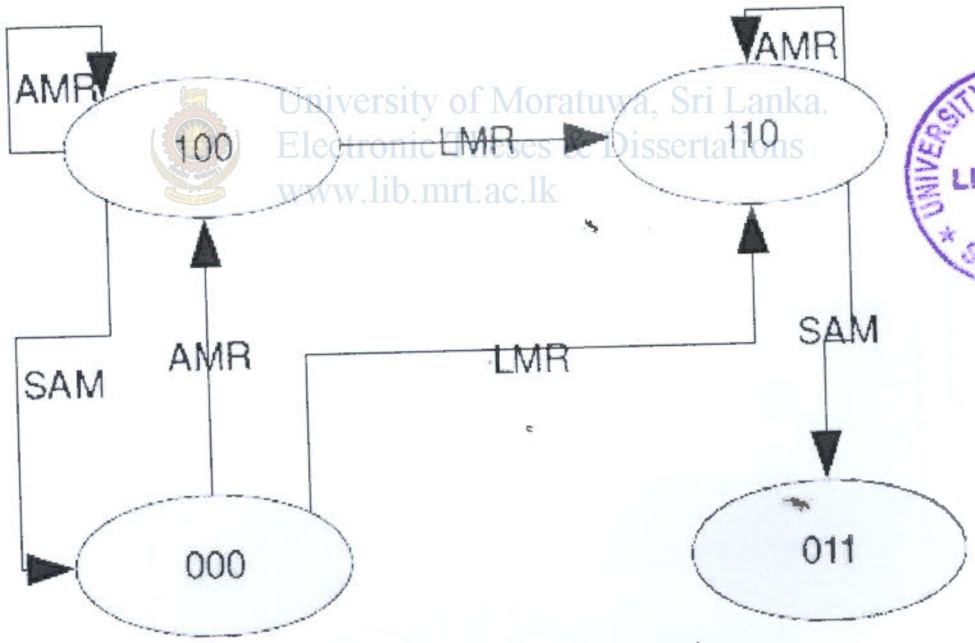


Figure 3-13 InvokerBuffer State Machine

3.4.2 Run time Architecture

WSO2 Mercury is a WS-RM implementation written on top of Apache Axis2. Apache Axis2 provides a set of extension points called handlers. These handlers forms the Axis2 Engine execution chain and can be deployed as modules. Therefore WSO2 Mercury in other words is an Axis2 module. A typical Axis2 message send starts with the application client which calls the service client. Then the message is passed through the Axis2 Engine handlers and finally is sent to the network using the transport sender. At the server side message is received by the transport receiver. After that it invokes the Axis2 Engine where message is passed through a set of handlers and finally receives at the message receiver which invokes the application at the server side.

WSO2 Mercury consists mainly of two handlers called MercuryOutHandler and MercuryInHandler which are used at out and in flows respectively, and a set of workers called RMSSequenceWorker, RMDSequenceWorker and InvokerBufferWorker which read the respective storages and perform the appropriate action based on the state.

Next set of scenarios shows the runtime architecture necessarily with the persistence storage which uses transactions to read/update storage. Although in memory storage does not support transactions it is also has the same runtime architecture.

In Only Messages

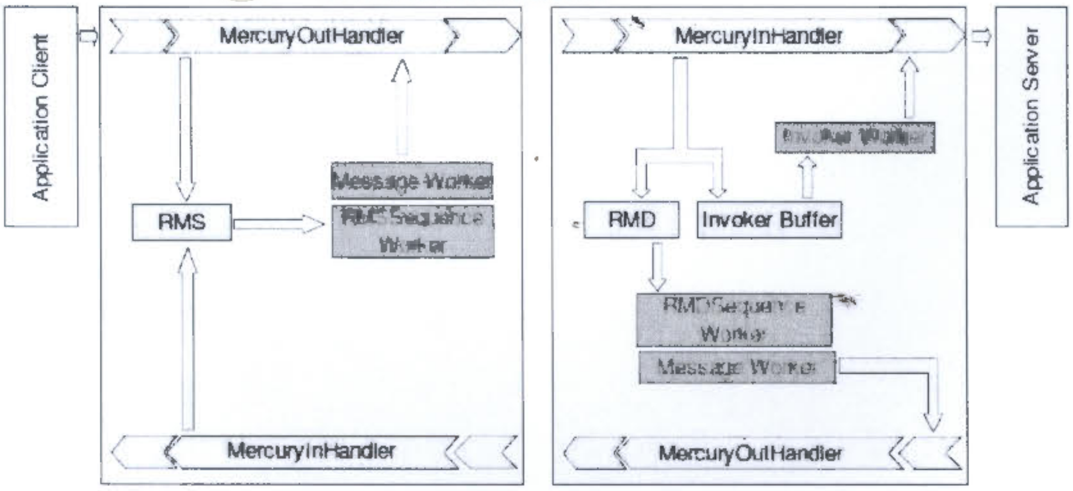


Figure 3-14 In Only Messages Runtime

Fig 3-14 shows the runtime execution for an in only message scenario. Mercury receives the message from the Application client at the MercuryOutHandler, creates a sequence if it already not there and stored the message in the RMS. RMSSequenceWorker picks this message and invokes the MessageWorker. MessageWorker sends this message through the rest of the handlers and finally the message being sent to the network through transport sender. At the server side transport receiver gets this message and invokes the Axis2 Engine. Mercury receives the message at the MercuryInHandler which updates the RMD and stores the message in the InvokerBuffer. Then InvokerBufferWorker picks this message from the InvokerBuffer and invokes the rest of the handlers so that ultimately message receives at the application at the server side.

Reliability of the WS-RM protocol is achieved by retransmissions and acknowledgments as in any other reliable protocol. A separate worker called RMDSequenceWorker is used to send acknowledgments back to the client side. Upon receiving an acknowledgment client side updates its' state as message has successfully send. As shown in the figure, RMDSequenceWorker reads the RMD state and sends an acknowledgment message using MessageWorker which generally is used to send any message. At the client side Mercury picks this message using the MercuryInHandler and it updates the RMS.

Although it is not shown in the diagram RMSSequenceWorker sends the create sequence message when establishing the sequence and sends the terminate sequence message to terminate the sequence. Similarly RMDSequenceWorker sends the create sequence response message according to the state of the RMD.

How this architecture supports user transactions? As it is shown in the Fig 3-14 it does not keep anything in memory. Any event reads and updates the storages using a transaction which is at the serializable isolation level. Therefore any update is not visible to other threads or workers until the transaction commits successfully.

In Out Messages

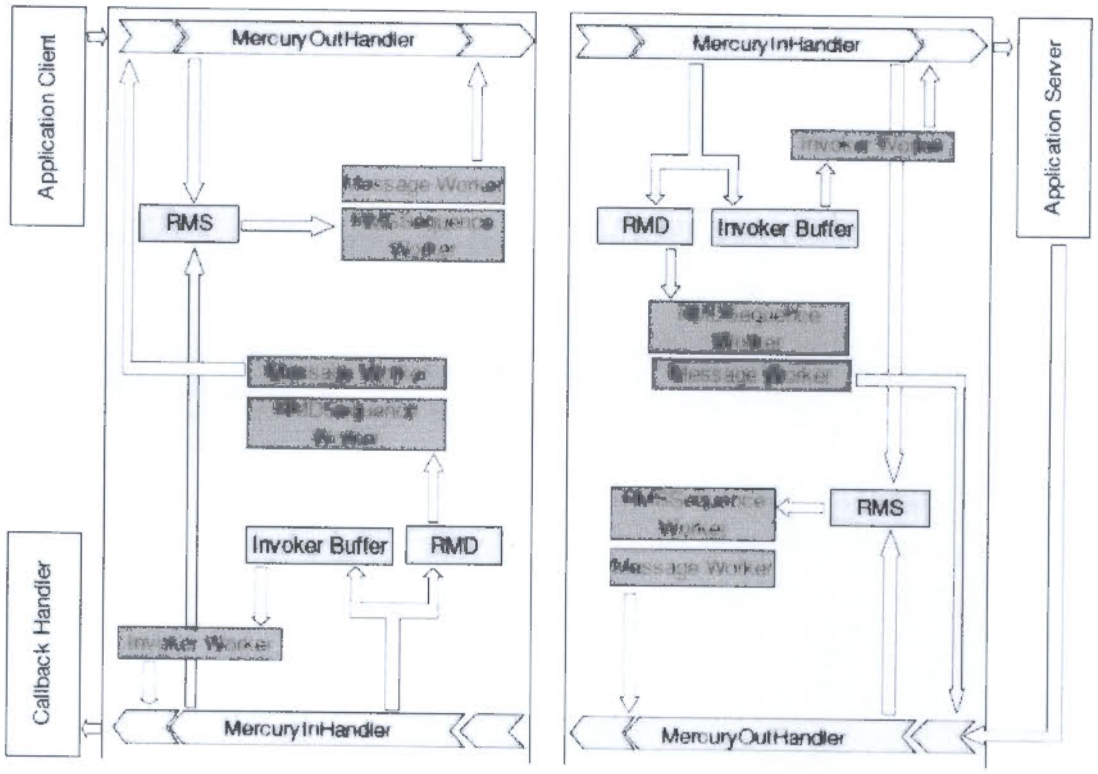


Figure 3-15 InOut Messages Runtime

Fig 3-15 shows the run time architecture for an in out scenario. The response path is similar to request path where the message receiver at the server side initiates the response message flow and it is ultimately received by the Axis2 callback. This axis callback is registered by the application client when sending the message.

Fault Handling

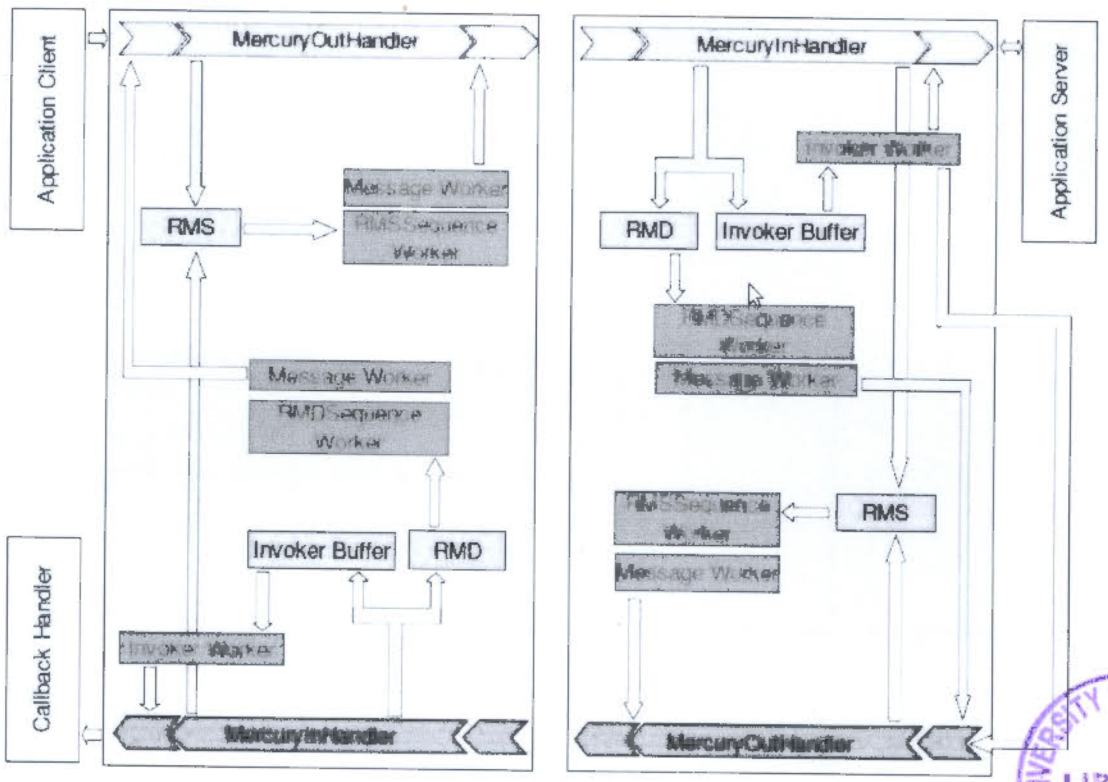


Figure 3-16 Fault Handling Runtime

Fig 3-16 shows the run time architecture for fault handling. In a fault scenario message receiver throws an AxisFault which Mercury takes as an application fault. This exception is captured at the InvokerBufferWorker level and it first roll backs the original transaction used to invoke the application. After that InvokerBufferWoker starts another transaction and sends the message using fault out flow. Client side scenario is similar except the message is received at the in fault flow.

3.4.3 Storage API

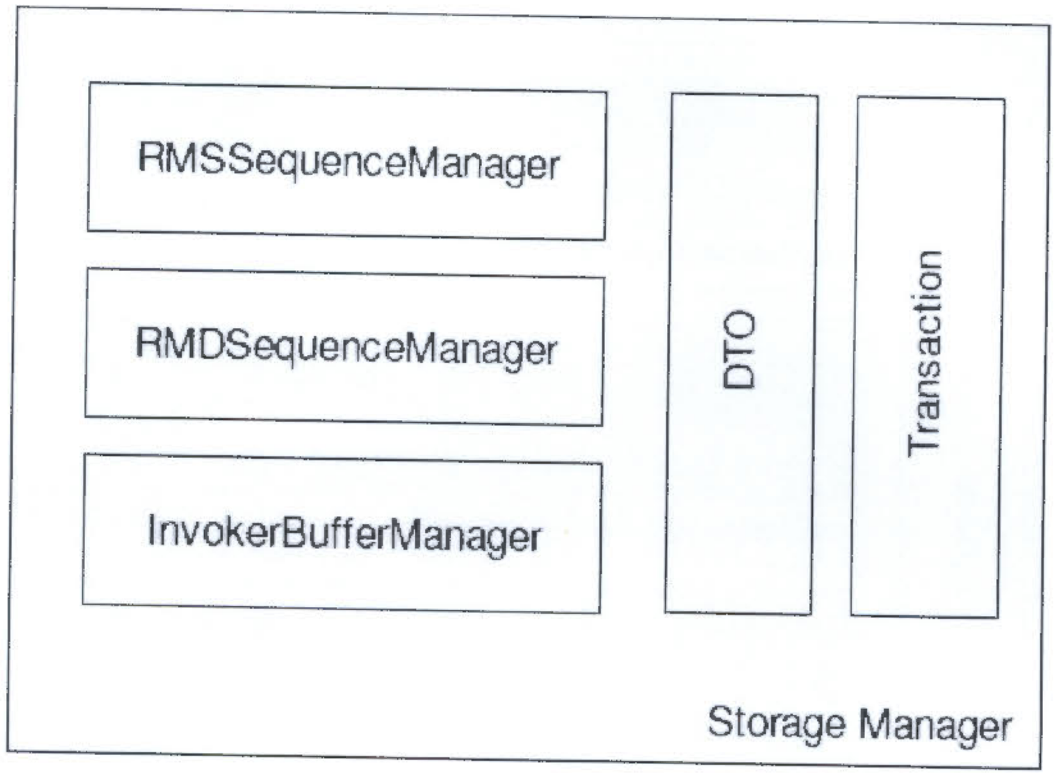


Figure 3-17 Storage API

Storage API mainly consists of a set of Manager interfaces namely RMSSequenceManager, RMDSequenceManager and InvokerBufferManager, Data transfer classes, Transaction interface to handle transactions and a top level StorageManager Interface which provides the access to other interfaces. StorageManager Interface provides the methods to get transactions and manager objects which provides the methods to manage respective storages. Before accessing the manager objects, the accessing thread should start a transaction by getting a transaction from the storage.

This storage API provides the explicit support to implement in memory and persistence storages in different ways. For an in memory storage, there is one set of manager objects for each sequence. The manager object for a particular sequence can be found using the parameters being passed to manager object access method. On the other hand for an persistence storage there can be one set of manager object for each sequence. In this case the correct storage dto object for a particular sequence is determined by the parameters passed to storage dto access method in the manager interface.

InMemory Implementation

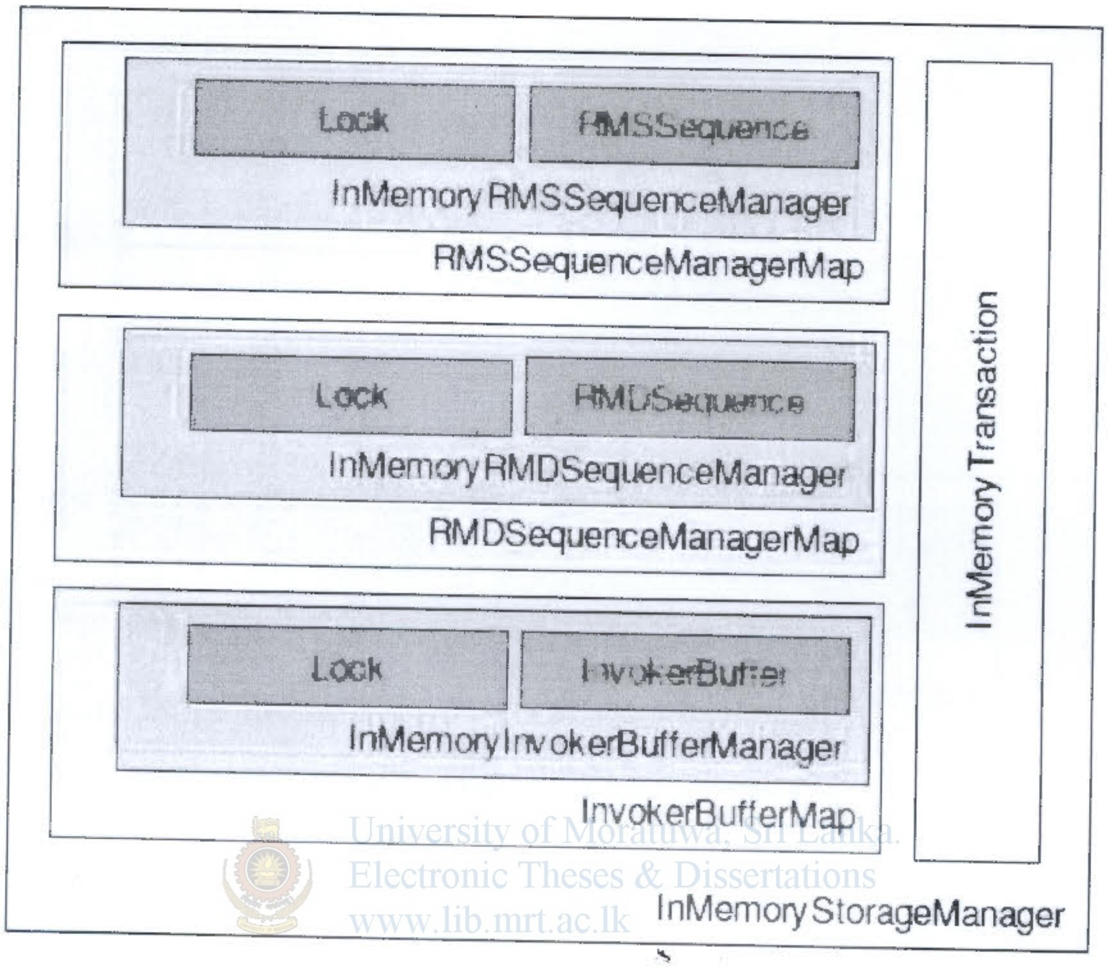


Figure 3-18 InMemory Implementation

Fig 3-18 shows the in memory storage design for Mercury. In memory storage keeps a separate sequence manager object for each sequence and it keeps these objects in three hash maps called iSKRMSequenceManagerMap, sequenceIDRMDSequenceMangerMap, sequenceIDInvokerBufferMap. A sequence manager object has a lock and another object to keep the details for the sequence manager object. Any sequence manager object can be retrieved from hash tables giving the key as the parameter. But before accessing the sequence manager object the corresponding transaction has to acquire the lock for that object.

Synchronization

For proper state machine execution only one thread can update the sequence at a given time. Hence it is required to synchronize the state machine or sequence manager objects. Two phase locking is used to synchronize the sequence manager objects where a transaction acquires the locks when accessing objects and releases them upon a commit or a rollback. A transaction always acquires sequence managers in the order of RMDSequenceManager, InvokerBufferManager and RMSSequenceManager to avoid deadlocks.

Persistence Implementation

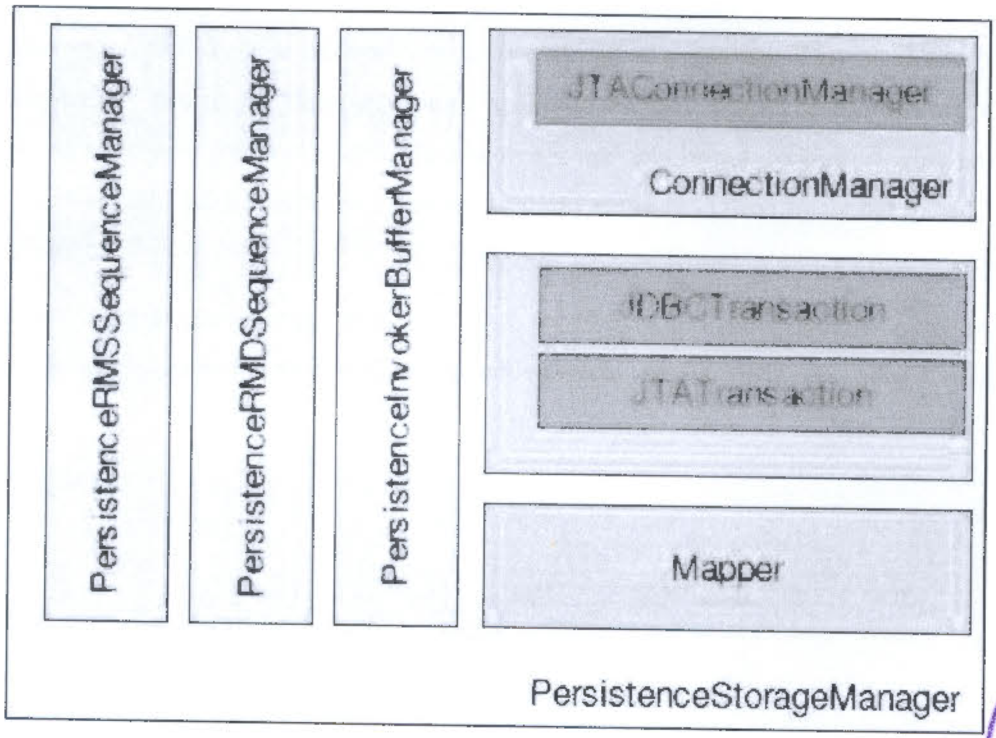

PersistenceRMSSequenceManager
PersistenceRMDSequenceManager
PersistenceInvokerBufferManager
JTAConnectionManager
ConnectionManager
JDBCTransaction
JTATransaction
Mapper
PersistenceStorageManager

Figure 3-19 Persistence Implementation

Fig 3-19 shows the important components of the persistence storage. It has a connection manager which is used to create either normal database connections or xaConnections to database. There are two types of transactions called JDBCTransactions and JTAThransaction. A JDBCTransaction contains a normal database connection where as a JTATransaction contains an xaConnections. Once a thread requests a transaction persistence storage access the connection manager and creates the requested type transaction. Unlike in the in memory model, persistence storage manager keeps one set of sequence manager objects for all sequences. All sequence managers use a set of helper classes called table mappers to create sql queries for dto objects and to create dto objects from result set objects. Sequence manager objects gets the connection object to use from the thread local.

Synchronization

Again for proper state machine execution only one thread hence a transaction can update the sequence state. This can be achieved by setting the isolation level of the transactions to TRANSACTION_SERIALIZABLE . This isolation level can leads to deadlocks.

First there can be deadlocks due to different order of table access. This has been solved by always accessing the RMDSequenceManager related tables first, then InvokerBufferManager related tables and finally RMSSequenceManager related tables. One transaction may not acquire all sequence manager objects but if it requires it has to access in the given order.

At TRANSACTION_SERIALIZABLE isolation level a transaction has to acquire a writer lock (an exclusive lock) to update a table. A writer lock can only be acquired after all the other transactions release the reader lock at a commit or a rollback. This gives another type of dead lock if two transactions try to read and update a table concurrently. Since both can not release the reader lock until write. This problem can only be solved by acquiring an exclusive lock at a read. An exclusive lock can be acquired at the read time by using 'select for update' statement.

Mercury persistence storage uses above two techniques to achieve synchronization avoiding deadlocks. It has been tested with an embedded Derby database with row level locking.

Database design

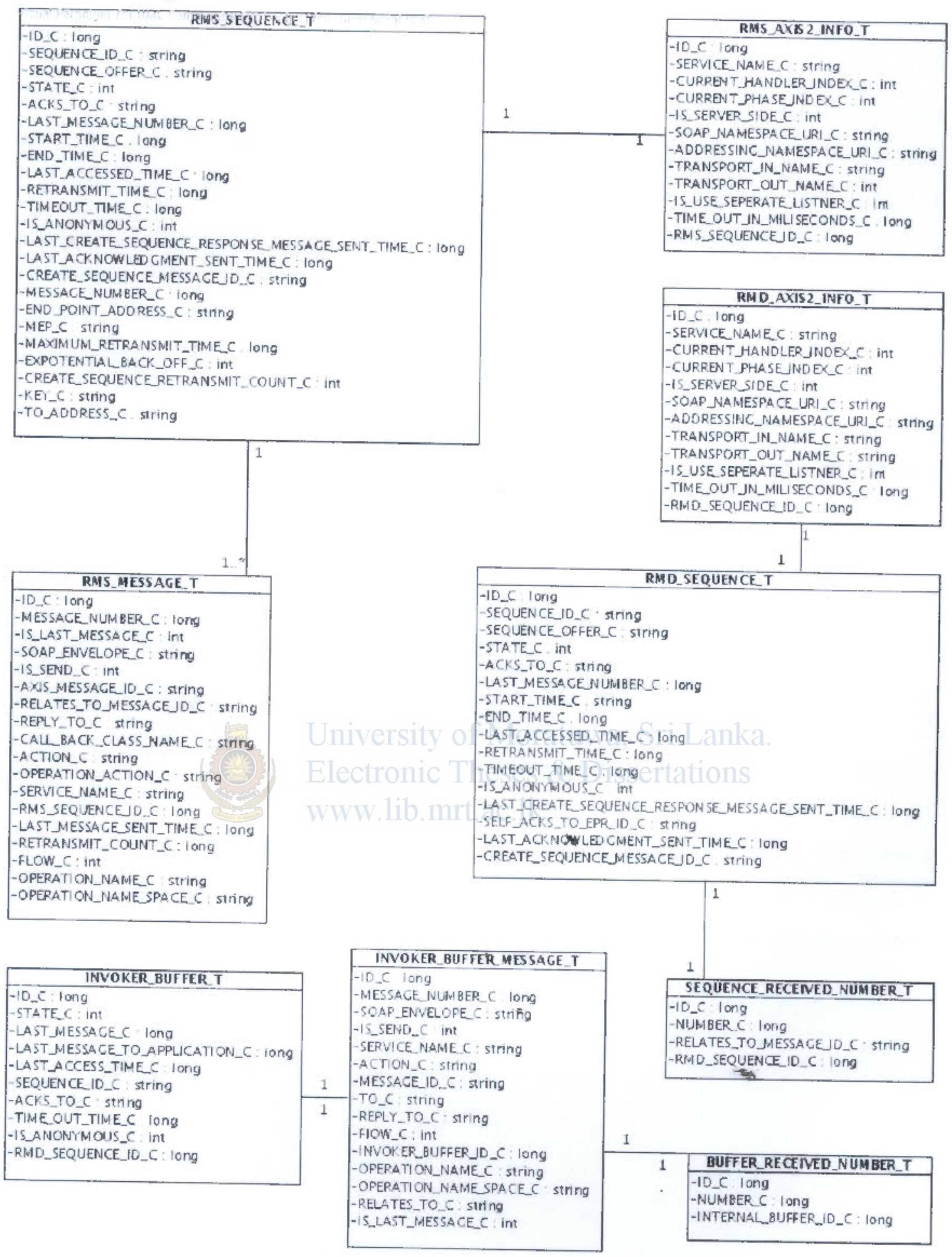

RMS_SEQUENCE_T
-ID_C : long
-SEQUENCE_ID_C : string
-SEQUENCE_OFFER_C : string
-STATE_C : int
-ACKS_TO_C : string
-LAST_MESSAGE_NUMBER_C : long
-START_TIME_C : long
-END_TIME_C : long
-LAST_ACCESSED_TIME_C : long
-RETRANSMIT_TIME_C : long
-TIMEOUT_TIME_C : long
-IS_ANONYMOUS_C : int
-LAST_CREATE_SEQUENCE_RESPONSE_MESSAGE_SENT_TIME_C : long
-LAST_ACKNOWLEDGMENT_SENT_TIME_C : long
-CREATE_SEQUENCE_MESSAGE_ID_C : string
-MESSAGE_NUMBER_C : long
-END_POINT_ADDRESS_C : string
-MEP_C : string
-MAXIMUM_RETRANSMIT_TIME_C : long
-EXPOTENTIAL_BACK_OFF_C : int
-CREATE_SEQUENCE_RETRANSMIT_COUNT_C : int
-KEY_C : string
-TO_ADDRESS_C : string
RMS_AXIS2_INFO_T
-ID_C : long
-SERVICE_NAME_C : string
-CURRENT_HANDLER_INDEX_C : int
-CURRENT_PHASE_INDEX_C : int
-IS_SERVER_SIDE_C : int
-SOAP_NAMESPACE_URI_C : string
-ADDRESSING_NAMESPACE_URI_C : string
-TRANSPORT_IN_NAME_C : string
-TRANSPORT_OUT_NAME_C : int
-IS_USE_SEPERATE_LISTNER_C : int
-TIME_OUT_IN_MILISECONDS_C : long
-RMS_SEQUENCE_ID_C : long
RMD_AXIS2_INFO_T
-ID_C : long
-SERVICE_NAME_C : string
-CURRENT_HANDLER_INDEX_C : int
-CURRENT_PHASE_INDEX_C : int
-IS_SERVER_SIDE_C : int
-SOAP_NAMESPACE_URI_C : string
-ADDRESSING_NAMESPACE_URI_C : string
-TRANSPORT_IN_NAME_C : string
-TRANSPORT_OUT_NAME_C : string
-IS_USE_SEPERATE_LISTNER_C : int
-TIME_OUT_IN_MILISECONDS_C : long
-RMD_SEQUENCE_ID_C : long
RMS_MESSAGE_T
-ID_C : long
-MESSAGE_NUMBER_C : long
-IS_LAST_MESSAGE_C : int
-SOAP_ENVELOPE_C : string
-IS_SEND_C : int
-AXIS_MESSAGE_ID_C : string
-RELATES_TO_MESSAGE_ID_C : string
-REPLY_TO_C : string
-CALL_BACK_CLASS_NAME_C : string
-ACTION_C : string
-OPERATION_ACTION_C : string
-SERVICE_NAME_C : string
-RMS_SEQUENCE_ID_C : long
-LAST_MESSAGE_SENT_TIME_C : long
-RETRANSMIT_COUNT_C : long
-FLOW_C : int
-OPERATION_NAME_C : string
-OPERATION_NAME_SPACE_C : string
RMD_SEQUENCE_T
-ID_C : long
-SEQUENCE_ID_C : string
-SEQUENCE_OFFER_C : string
-STATE_C : int
-ACKS_TO_C : string
-LAST_MESSAGE_NUMBER_C : long
-START_TIME_C : string
-END_TIME_C : long
-LAST_ACCESSED_TIME_C : long
-RETRANSMIT_TIME_C : long
-TIMEOUT_TIME_C : long
-IS_ANONYMOUS_C : int
-LAST_CREATE_SEQUENCE_RESPONSE_MESSAGE_SENT_TIME_C : long
-SELF_ACKS_TO_EPR_ID_C : string
-LAST_ACKNOWLEDGMENT_SENT_TIME_C : long
-CREATE_SEQUENCE_MESSAGE_ID_C : string
INVOKER_BUFFER_T
-ID_C : long
-STATE_C : int
-LAST_MESSAGE_C : long
-LAST_MESSAGE_TO_APPLICATION_C : long
-LAST_ACCESS_TIME_C : long
-SEQUENCE_ID_C : string
-ACKS_TO_C : string
-TIME_OUT_TIME_C : long
-IS_ANONYMOUS_C : int
-RMD_SEQUENCE_ID_C : long
INVOKER_BUFFER_MESSAGE_T
-ID_C : long
-MESSAGE_NUMBER_C : long
-SOAP_ENVELOPE_C : string
-IS_SEND_C : int
-SERVICE_NAME_C : string
-ACTION_C : string
-MESSAGE_ID_C : string
-TO_C : string
-REPLY_TO_C : string
-FLOW_C : int
-INVOKER_BUFFER_ID_C : long
-OPERATION_NAME_C : string
-OPERATION_NAME_SPACE_C : string
-RELATES_TO_C : string
-IS_LAST_MESSAGE_C : int
SEQUENCE_RECEIVED_NUMBER_T
-ID_C : long
-NUMBER_C : long
-RELATES_TO_MESSAGE_ID_C : string
-RMD_SEQUENCE_ID_C : long
BUFFER_RECEIVED_NUMBER_T
-ID_C : long
-NUMBER_C : long
-INTERNAL_BUFFER_ID_C : long
1
1..*

Figure 3-20 Database Design

Fig 3-20 shows the underline database design for persistence storage. It contains a separate set of tables for each sequence manager in order to avoid deadlocks by accessing them in a clear order always.

3.4.4 Other issues and solutions

Starting the terminated sequences at the client side

Client node can fail while sending a sequence of messages. Therefore for application client there is no way to know whether it properly terminated the sequence or not if the client node fails just after sending the all the messages (this case happens only when there is no explicit last message but application client sends a terminate message to Mercury). As a solution to this problem Mercury sends an explicit terminate signal for all the sequences that has not been terminated. If the application client has not send all the messages then it can start a new sequence and send the remaining messages.

For in out client scenarios once the client node fails, addressing dispatch information stored at configuration context also get lost. And also there is no axis2 service to receive the messages as well.

In order to solve the above two issues Mercury uses a deployment life cycle listener to terminate the non terminated sequences, to add the axis2 service and to register dispatch information in order to dispatch sequences.

Distributed transaction recovery

Two phase commit (2PC) protocol guarantees the atomicity of a global transaction even when node fails. 2PC protocol has a recovery phase to recover from the node failures if the coordinator or any other node fails within the commit phase. Therefore in order to guarantees the atomicity of the global transaction the XA implementations should properly support the recovery phase. However it seems some database XA drivers have problems with the recovery phase.

Chapter 4

Use case scenarios

Mercury can be used to invoke services using both in only and in out message exchange patterns (MEP). Although this research work focus on user level transaction support it is designed in a way that it can be used with simple inmemory implementations as well. Following use case scenarios are used to demonstrate how to use Mercury with different storage implementation types. WS-RM 1.0 describes an addressing based dual channel mode to send and receive messages. Therefore for all use case scenarios given here uses addressing based dual channel mode. There is another specification describes an piggyback message based system which uses http back channel to receive messages. Mercury supports the latter kind of invocations only with the in memory implementations.

4.1 InMemory

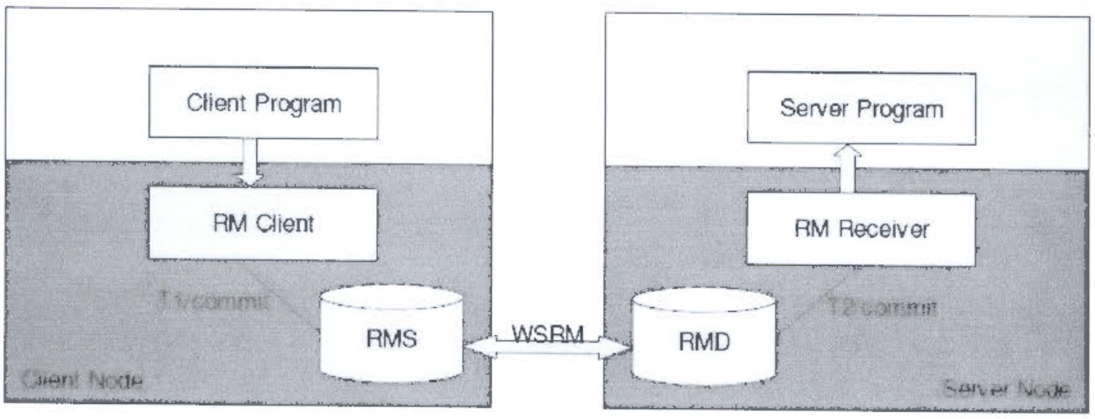


Figure 4-1 InMemory Invocation

InMemory invocation is the most simple way of using Mercury. It does not requires to do anything other than the engaging the Mercury module as in any other module engagement. Mercury uses in memory implementation as the default storage. Messages can be send through a tcp monitor and start and stop channels in order to prove the reliability with the presence of network failures.

4.1.1 In Only invocation

```
ConfigurationContext configurationContext =
        ConfigurationContextFactory.createConfigurationContextFromFileSystem(
                AXIS2_REPOSITORY_LOCATION, AXIS2_CLIENT_CONFIG_FILE);
ServiceClient serviceClient = new ServiceClient(configurationContext, null);
serviceClient.setTargetEPR(new EndpointReference("http://localhost:8088/axis2/services/InMemoryInService"));
serviceClient.getOptions().setAction("urn:InMemoryInOperation");
serviceClient.engageModule("Mercury");
serviceClient.getOptions().setUseSeparateListener(true);
serviceClient.getOptions().setProperty(MercuryClientConstants.INTERNAL_KEY, "key1");
for (int i = 1; i < 20; i++) {
    serviceClient.fireAndForget(getTestOMElement(i));
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
    }
}
MercuryClient mercuryClient = new MercuryClient(serviceClient);
mercuryClient.terminateSequence("key1");
```

Figure 4-2 In Only Client

It uses a service client object to invoke the service. First it creates a configuration context pointing to an repository location. Then it sets the endpoint reference and soap action associated with the operation as in any Axis2 client invocation. After that it engage Mercury module in order to make this connection reliable. Here InMemoryInService should also have engaged the Mercury module. It sets the useSeperateListner parameter to make this a dual channel invocation. Mercury uses the internal key parameter to distinguish messages belonging to different sequences. After setting all the necessary parameters it sends 20 messages and finally terminate the sequence by invoking the terminate sequence method.

4.1.2 In Out Invocation

```
ConfigurationContext configurationContext =
        ConfigurationContextFactory.createConfigurationContextFromFileSystem(
                AXIS2_REPOSITORY_LOCATION, AXIS2_CLIENT_CONFIG_FILE);
ServiceClient serviceClient = new ServiceClient(configurationContext, null);
serviceClient.setTargetEPR(new EndpointReference("http://localhost:8088/axis2/services/InMemoryInOutService"));
serviceClient.getOptions().setAction("urn:InMemoryInOutOperation");
serviceClient.getOptions().setUseSeparateListener(true);

serviceClient.getOptions().setProperty(MercuryClientConstants.SEQUENCE_OFFER, Constants.VALUE_TRUE);
serviceClient.engageModule("Mercury");
for (int i = 1; i < 20; i++) {
    sendAsynchornousMessage(serviceClient, i, "Key1");
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
    }
}

MercuryClient mercuryClient = new MercuryClient(serviceClient);
mercuryClient.terminateSequence("Key1");
```

Figure 4-3 InMemory In Out Client

WS-RM supports in out invocations by establishing two RM sequences for in and out message sequences. For incoming sequence an sequence identifier can be offered when sending the createSequence message for out sequence. In this sample client it sets the sequence offer to ask Mercury to send a sequence offer with the createSeqence message. Unlike in the in only scenario it does an asynchronous in out invocation using the sendAsynchronousMessage method.

```
private void sendAsynchornousMessage(ServiceClient serviceClient, int i, String key) throws AxisFault {
    serviceClient.getOptions().setProperty(MercuryClientConstants.INTERNAL_KEY, key);
    AxisCallback axisCallback = new AxisCallback() {
        public void onMessage(MessageContext msgContext) {
            System.out.println("Got the message ==> " + msgContext.getEnvelope().getBody().getFirstElement());
        }

        public void onFault(MessageContext msgContext) {
            System.out.println("Got the fault ==> " + msgContext.getEnvelope().getBody().getFault().getDetail());
        }

        public void onError(Exception e) {
            e.printStackTrace();
        }

        public void onComplete() {

        }
    };
    serviceClient.sendReceiveNonBlocking(getTestOMElement(key + " " + i + " "), axisCallback);
}
```

Figure 4-4 SendAsynchornousMessage Method

It sets an Axis Call back object to receive the messages and do an asynchronous invocation so that it can send the out messages without waiting for the incoming sequence.

4.1.3 Fault Handling

```java
public class InMemoryFaultMessageReceiver extends AbstractInOutMessageReceiver {

    public void invokeBusinessLogic(MessageContext inMessage, MessageContext outMessage)
            throws AxisFault {
        System.out.println("Sending the fault message");
        AxisFault axisFault = new AxisFault("TestError message");
        axisFault.setDetail(getTestOMElement(inMessage.getEnvelope().getBody().getFirstElement().getText()));
        throw axisFault;

    }

    private OMElement getTestOMElement(String text){
        OMFactory omFactory = OMAbstractFactory.getOMFactory();
        OMNamespace omNamespace = omFactory.createOMNamespace("http://wso2.org/temp1","ns1");
        OMElement omElement = omFactory.createOMElement("TestErrorElement",omNamespace);
        omElement.setText("Reply " + text);
        return omElement;
    }
}
```

Figure 4-5 Fault Message Receiver

Fault scenarios has been implemented by using a message receiver which always sends an AxisFault. Mercury sends application level exceptions reliably by using the response message sequence. When an application exception receives at the InvokerWorker it rollbacks the transaction used to invoke the business logic and starts a new transaction to send the fault message.

4.2 Persistence

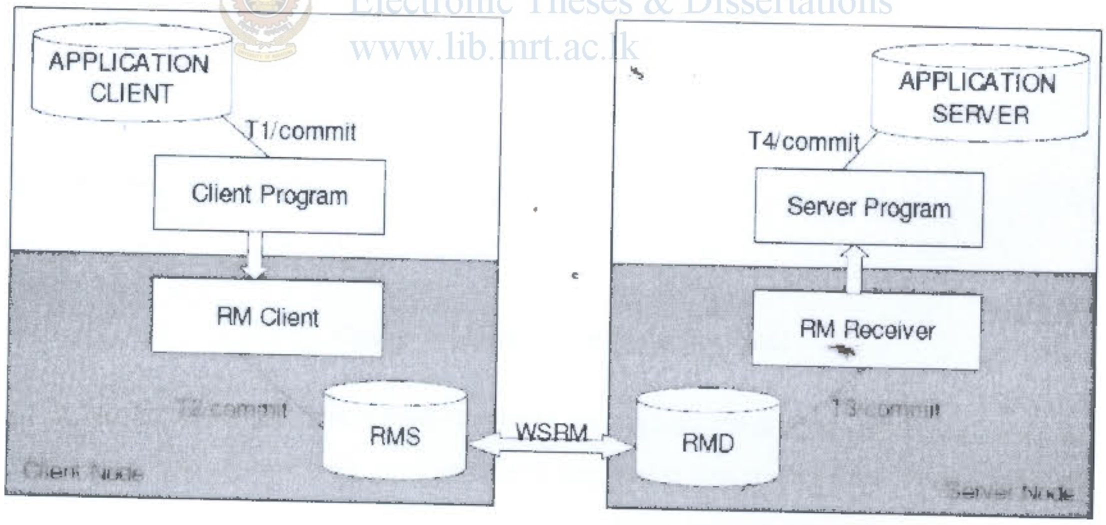


Figure 4-6 Using Persistence Storage

Fig 4-6 shows a sample persistence storage usage to transfer set of messages stored at the APPLICATION_CLIENT storage to APPLICATION_SERVER storage. This configuration works even with the presence of node failures (i.e. start and stop client or server) but there can be message losses if the node fails at the stages where message only resides in memory.

In order to use a persistence storage it has to be configured at the axis2.xml file both at the server and client side. We use Apache Derby as the underlying database.

```
<parameter name="storageManagerClass">org.wso2.mercury.storage.impl.persistence.PersistenceStorageMeanager</parameter>
<moduleConfig name="Mercury">
    <parameter name="db.connectionstring">jdbc:derby:/home/amila/msc/project/mercury/modules/demo/persistence/client/database/MERCURY_DB</parameter>
    <parameter name="db.driver">org.apache.derby.jdbc.EmbeddedDriver</parameter>
    <parameter name="db.user"></parameter>
    <parameter name="db.password"></parameter>
</moduleConfig>
```

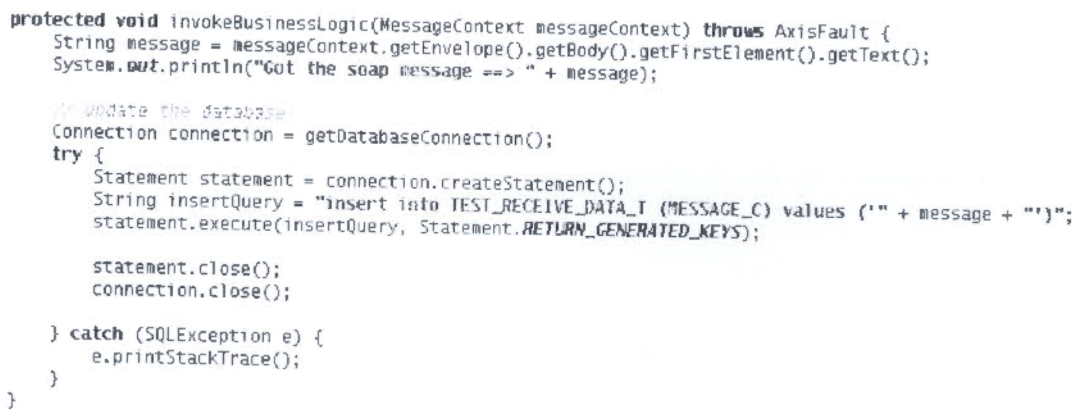
Figure 4-7 Persistence Storage Configuration

4.2.1 In Only Invocation

```
ConfigurationContext configurationContext =
        ConfigurationContextFactory.createConfigurationContextFromFileSystem(
                AXIS2_REPOSITORY_LOCATION, AXIS2_CLIENT_CONFIG_FILE);
ServiceClient serviceClient = new ServiceClient(configurationContext, null);
serviceClient.setTargetEPR(new EndpointReference("http://localhost:8088/axis2/services/PersistenceInService"));
serviceClient.getOptions().setAction("urn:PersistenceInOperation");
serviceClient.engageModule("Mercury");
serviceClient.getOptions().setUseSeparateListener(true);
serviceClient.getOptions().setProperty(MercuryClientConstants.INTERNAL_KEY, "key1");

//read the messages from the data base and send them
Connection connection = getDatabaseConnection();

Statement statement = connection.createStatement();
String sqlString = "select ID_C from TEST_SEND_DATA_T where IS_SEND_C=0";
ResultSet resultSet = statement.executeQuery(sqlString);
List<Long> messageNumbers = new ArrayList<Long>();
while (resultSet.next()) {
    messageNumbers.add(resultSet.getLong("ID_C"));

}
resultSet.close();

for (long messageID : messageNumbers) {
    String queryString = "select * from TEST_SEND_DATA_T where ID_C=" + messageID;
    resultSet = statement.executeQuery(queryString);
    if (resultSet.next()) {
        String message = resultSet.getString("MESSAGE_C");
        String updateString = "update TEST_SEND_DATA_T set IS_SEND_C=1 where ID_C=" + messageID;
        statement.executeUpdate(updateString);
        System.out.println("Sending message " + messageID);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        serviceClient.fireAndForget(getTestOMElement(message));
    }
}

statement.close();
connection.close();

MercuryClient mercuryClient = new MercuryClient(serviceClient);
mercuryClient.terminateSequence("key1");
```

Figure 4-8 Persistence In Only Client

As shown in the Fig 4-8 first it reads available message numbers to be send within a separate database connection. Then for each and every message it gets the message from the database record and updates message as send. Since we have not set the auto commit to false, executeUpdate statement commit the transaction automatically. After that as in the in memory case it sends the message. If the client node fails while it sleeps then this message get lost at the client side.



```
protected void invokeBusinessLogic(MessageContext messageContext) throws AxisFault {
    String message = messageContext.getEnvelope().getBody().getFirstElement().getText();
    System.out.println("Got the soap message ==> " + message);

    // update the database
    Connection connection = getDatabaseConnection();
    try {
        Statement statement = connection.createStatement();
        String insertQuery = "insert into TEST_RECEIVE_DATA_T (MESSAGE_C) values ('" + message + "')";
        statement.execute(insertQuery, Statement.RETURN_GENERATED_KEYS);

        statement.close();
        connection.close();

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Figure 4-9 Persistence In Only MessageReceiver

At the message receiver it saves the message to APPLICATION_SERVER database within a transaction. Before sending this message InvokerWorker reads the message from the RMD and starts a transaction but commit it after invoking the business logic. Therefore duplicate message can result if server node fail before invokerWorker commits the transaction.

4.2.2 In Out Invocation and Fault Handling

Both In Out and Fault handling clients are almost equal to the in memory cases but they read the messages from the database and update before sending them as given in the in only case. The only difference is that persistence invocations uses concrete AxisCallback class to receive the messages. This is useful when a client node restart while transmitting a sequence of messages. Then Mercury can create an instance of callback and register it at the Axisoperation callback receiver.

```
public void onMessage(MessageContext msgContext) {
    String message = msgContext.getEnvelope().getBody().getFirstElement().getText();
    System.out.println("OM Element ==> " + message);
    // update the database
    Connection connection = getDatabaseConnection();
    try {
        Statement statement = connection.createStatement();
        String insertQuery = "insert into TEST_RECEIVE_DATA_T (MESSAGE_C) values ('" + message + "')";
        statement.execute(insertQuery, Statement.RETURN_GENERATED_KEYS);

        statement.close();
        connection.close();

    } catch (SQLException e) {
        e.printStackTrace();
    }
}

public void onFault(MessageContext msgContext) {
    String message = msgContext.getEnvelope().getBody().getFault().getDetail().getFirstElement().getText();
    System.out.println("OM Element ==> " + message);

    // update the database
    Connection connection = getDatabaseConnection();
    try {
        Statement statement = connection.createStatement();
        String insertQuery = "insert into TEST_RECEIVE_DATA_T (MESSAGE_C) values ('" + message + "')";
        statement.execute(insertQuery, Statement.RETURN_GENERATED_KEYS);

        statement.close();
        connection.close();

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Figure 4-10 Persistence Callback Handler Methods

4.3 JTA

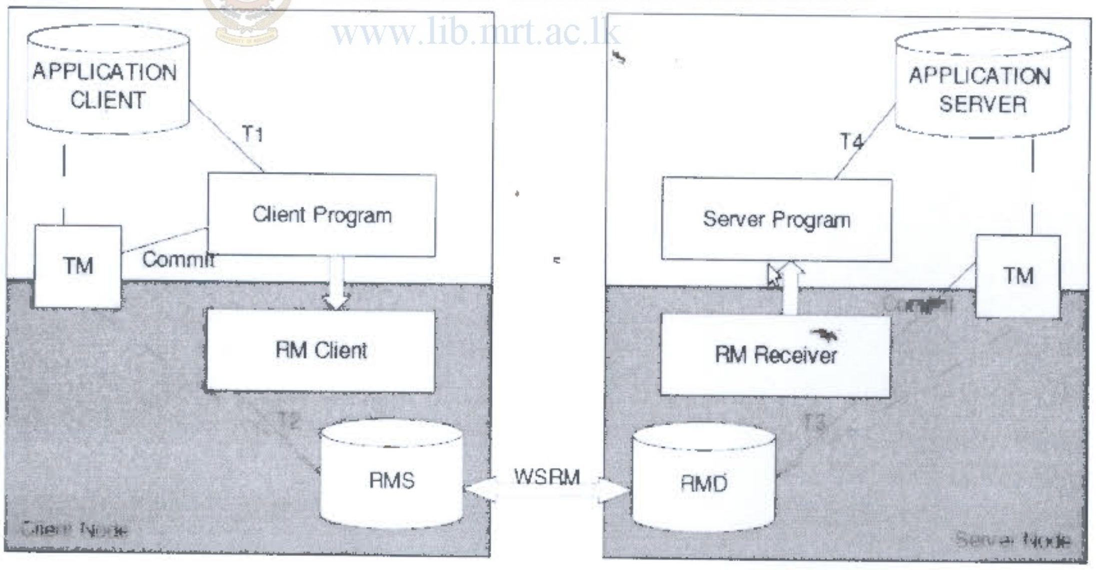
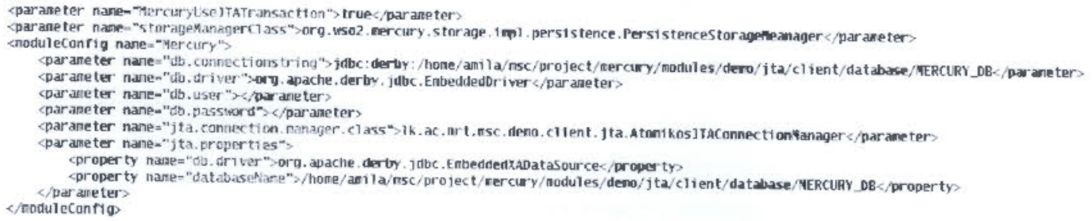


Figure 4-11 JTA Invocation

Fig 4-11 shows a sample JTA usage with Mercury to reliably transfer a set of messages stored at the APPLICATION_CLIENT to APLICATION_SERVER database. This configuration does not loose or send duplicate messages due to recovery nature of the distributed transactions. Here we use the Apache Derby as the underlying database and the Atomikos opensource library to provide the JTA functionality. The persistence storage with the jta connection manager have to be configured both at the client and server axis2.xml files.



```
<parameter name="MercuryUseJTATransaction">true</parameter>
<parameter name="storageManagerClass">org.wso2.mercury.storage.impl.persistence.PersistenceStorageMeanager</parameter>
<moduleConfig name="Mercury">
    <parameter name="db.connectionstring">jdbc:derby:/home/amila/msc/project/mercury/modules/demo/jta/client/database/MERCURY_DB</parameter>
    <parameter name="db.driver">org.apache.derby.jdbc.EmbeddedDriver</parameter>
    <parameter name="db.user"></parameter>
    <parameter name="db.password"></parameter>
    <parameter name="jta.connection.manager.class">lk.ac.mrt.msc.demo.client.jta.AtomikosJTAConnectionManager</parameter>
    <parameter name="jta.properties">
        <property name="db.driver">org.apache.derby.jdbc.EmbeddedXADataSource</property>
        <property name="databaseName">/home/amila/msc/project/mercury/modules/demo/jta/client/database/MERCURY_DB</property>
    </parameter>
</moduleConfig>
```

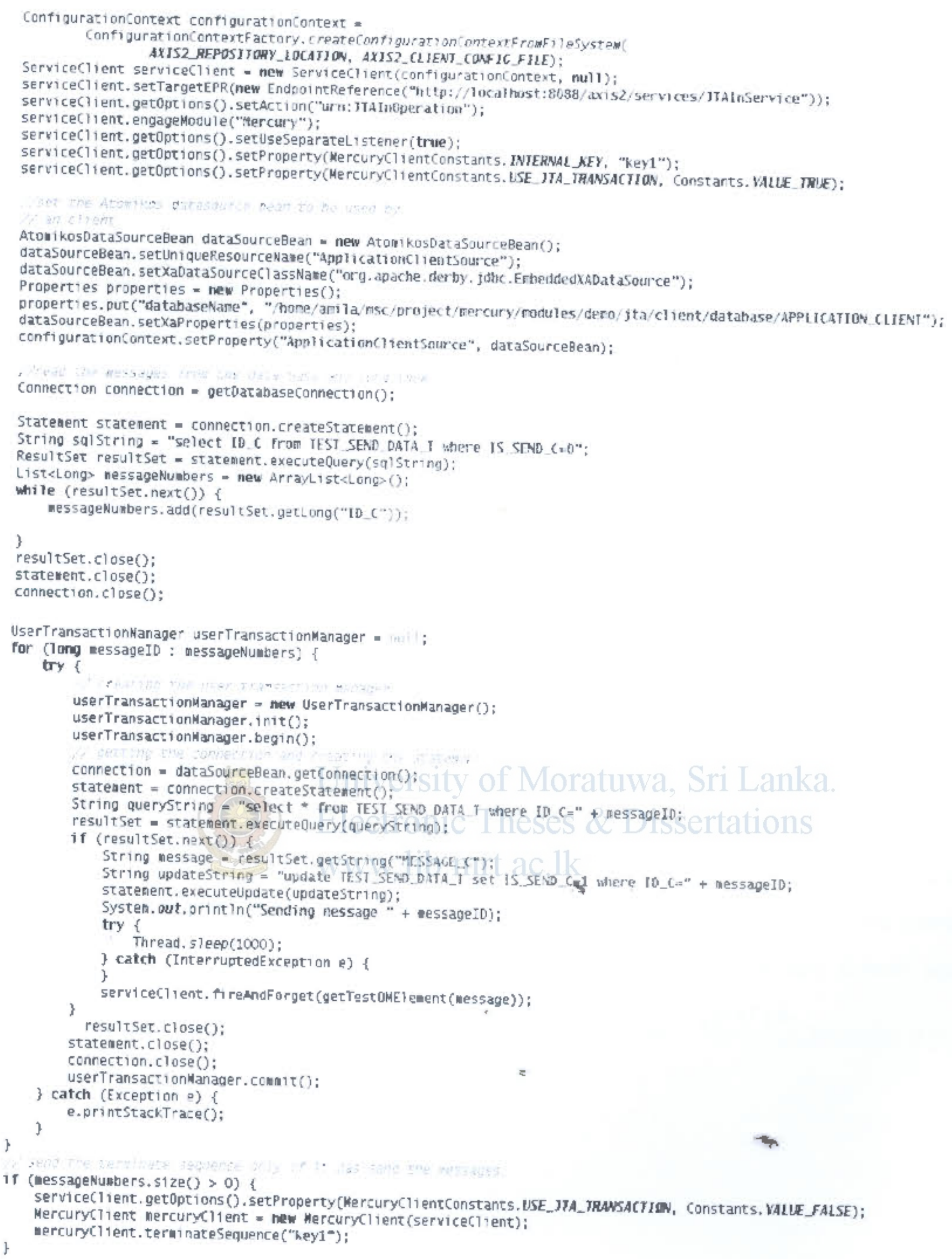
Figure 4-12 JTA Storage Configuration

For jta connections application client program supposed to provide a JTAConnectionManager which is used to get the transaction manager and connection objects. InvokerWorker use MercuryUseJTATransaction property to decide whether to start a JTA transaction or not before invoking the business logic.

```
public class AtomikosJTAConnectionManager implements JTAConnectionManager {

    private static Log log = LogFactory.getLog(AtomikosJTAConnectionManager.class);

    private String dbDriver = null;
    private Properties properties = null;
    private AtomikosDataSourceBean dataSourceBean = null;

    public void init(OMElement jtaPropertiesElement) throws StorageException {

        OMElement omElement = null;
        String propertyName = null;
        this.properties = new Properties();
        for (Iterator<OMElement> iter = jtaPropertiesElement.getChildElements(); iter.hasNext();) {
            omElement = iter.next();
            propertyName = omElement.getAttributeValue(new QName("", "name"));
            if (propertyName.equals(Constants.DB_DRIVER)) {
                this.dbDriver = omElement.getText();
            } else {
                properties.put(propertyName, omElement.getText());
            }
        }

        this.dataSourceBean = new AtomikosDataSourceBean();
        this.dataSourceBean.setUniqueResourceName("MercuryDataSource");
        this.dataSourceBean.setXaDataSourceClassName(dbDriver);
        this.dataSourceBean.setXaProperties(this.properties);
        this.dataSourceBean.setMaxPoolSize(5);

    }

    public Connection getNewConnection() throws StorageException {

        try {
            Connection connection = this.dataSourceBean.getConnection();
            return connection;
        } catch (SQLException e) {
            log.error("Can not create the Atomikos connection", e);
            throw new StorageException("Can not create the Atomikos connection", e);
        }
    }

    public TransactionManager getTransactionManager() throws StorageException {
        try {
            UserTransactionManager userTransactionManager = new UserTransactionManager();
            userTransactionManager.init();
            return userTransactionManager;
        } catch (SystemException e) {
            throw new StorageException("Can not init the Transaction manager");
        }
    }

}
```

Figure 4-13 Atomikos JTA Connection Manager

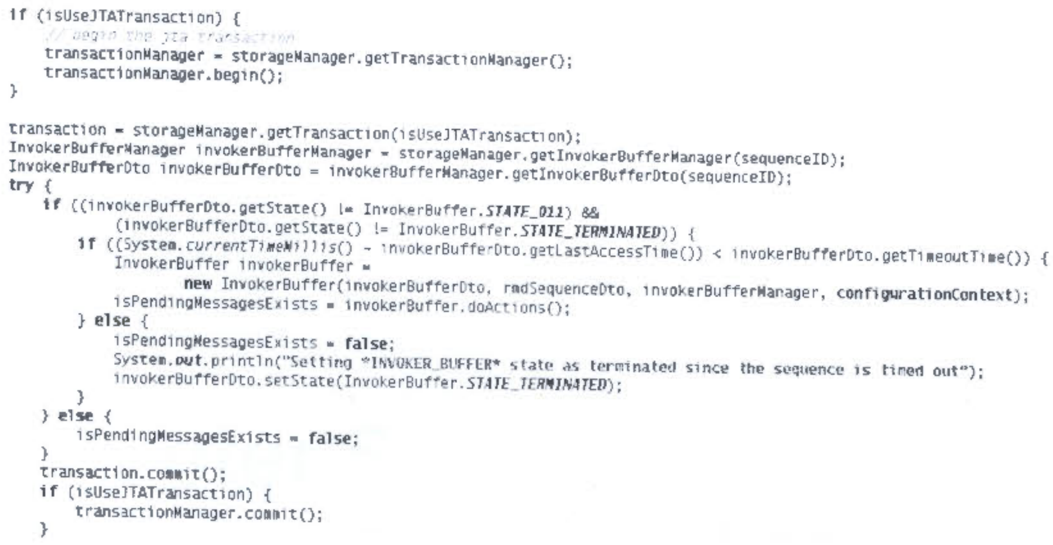
4.3.1 In Only Invocation



```
ConfigurationContext configurationContext =
        ConfigurationContextFactory.createConfigurationContextFromFileSystem(
                AXIS2_REPOSITORY_LOCATION, AXIS2_CLIENT_CONFIG_FILE);
ServiceClient serviceClient = new ServiceClient(configurationContext, null);
serviceClient.setTargetEPR(new EndpointReference("http://localhost:8088/axis2/services/JTAInService"));
serviceClient.getOptions().setAction("urn:JTAInOperation");
serviceClient.engageModule("Mercury");
serviceClient.getOptions().setUseSeparateListener(true);
serviceClient.getOptions().setProperty(MercuryClientConstants.INTERNAL_KEY, "key1");
serviceClient.getOptions().setProperty(MercuryClientConstants.USE_JTA_TRANSACTION, Constants.VALUE_TRUE);

//set the Atomikos datasource bean to be used by
// an client
AtomikosDataSourceBean dataSourceBean = new AtomikosDataSourceBean();
dataSourceBean.setUniqueResourceName("ApplicationClientSource");
dataSourceBean.setXaDataSourceClassName("org.apache.derby.jdbc.EmbeddedXADataSource");
Properties properties = new Properties();
properties.put("databaseName", "/home/amila/msc/project/mercury/modules/demo/jta/client/database/APPLICATION_CLIENT");
dataSourceBean.setXaProperties(properties);
configurationContext.setProperty("ApplicationClientSource", dataSourceBean);

//read the messages from the data base and send them
Connection connection = getDatabaseConnection();

Statement statement = connection.createStatement();
String sqlString = "select ID_C from TEST_SEND_DATA_T where IS_SEND_C=0";
ResultSet resultSet = statement.executeQuery(sqlString);
List<Long> messageNumbers = new ArrayList<Long>();
while (resultSet.next()) {
    messageNumbers.add(resultSet.getLong("ID_C"));

}
resultSet.close();
statement.close();
connection.close();


UserTransactionManager userTransactionManager = null;
for (long messageID : messageNumbers) {
    try {
        // creating the user transaction manager
        userTransactionManager = new UserTransactionManager();
        userTransactionManager.init();
        userTransactionManager.begin();
        // getting the connection and creating the statement
        connection = dataSourceBean.getConnection();
        statement = connection.createStatement();
        String queryString = "select * from TEST_SEND_DATA_T where ID_C=" + messageID;
        resultSet = statement.executeQuery(queryString);
        if (resultSet.next()) {
            String message = resultSet.getString("MESSAGE_C");
            String updateString = "update TEST_SEND_DATA_T set IS_SEND_C=1 where ID_C=" + messageID;
            statement.executeUpdate(updateString);
            System.out.println("Sending message " + messageID);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
            serviceClient.fireAndForget(getTestOMElement(message));
        }
          resultSet.close();
        statement.close();
        connection.close();
        userTransactionManager.commit();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
// send the terminate sequence only if it has send the messages.
if (messageNumbers.size() > 0) {
    serviceClient.getOptions().setProperty(MercuryClientConstants.USE_JTA_TRANSACTION, Constants.VALUE_FALSE);
    MercuryClient mercuryClient = new MercuryClient(serviceClient);
    mercuryClient.terminateSequence("key1");
}
```

Figure 4-14 JTA In Only Client

First it sets the USE_JTA_TRANSACTION property to true in order to indicate application client going to perform a jta transaction. When the Mercury sees this property it invokes the AtomikosJTAConnectionManager class to obtain the JTA connection resource. Then it creates an AtomikosDataSourceBean to get the JTA connection resource. AtomikosDataSourceBean automatically enlist the connection to distributed transaction. After that it reads the available message numbers to send using a normal database connection. Next it starts sending messages one by one. Unlike in a normal persistence scenario now it starts a distributed transaction using UserTransactionManager. After updating the client database and sending message to Mercury it closes the connection and commit the distributed transaction. Hence it reliably transfer messages stored at APPLICATION_CLIENT database to RMS.



```
if (isUseJTATransaction) {
    // begin the jta transaction
    transactionManager = storageManager.getTransactionManager();
    transactionManager.begin();
}

transaction = storageManager.getTransaction(isUseJTATransaction);
InvokerBufferManager invokerBufferManager = storageManager.getInvokerBufferManager(sequenceID);
InvokerBufferDto invokerBufferDto = invokerBufferManager.getInvokerBufferDto(sequenceID);
try {
    if ((invokerBufferDto.getState() != InvokerBuffer.STATE_011) &&
            (invokerBufferDto.getState() != InvokerBuffer.STATE_TERMINATED)) {
        if ((System.currentTimeMillis() - invokerBufferDto.getLastAccessTime()) < invokerBufferDto.getTimeoutTime()) {
            InvokerBuffer invokerBuffer =
                    new InvokerBuffer(invokerBufferDto, rmdSequenceDto, invokerBufferManager, configurationContext);
            isPendingMessagesExists = invokerBuffer.doActions();
        } else {
            isPendingMessagesExists = false;
            System.out.println("Setting *INVOKER_BUFFER* state as terminated since the sequence is timed out");
            invokerBufferDto.setState(InvokerBuffer.STATE_TERMINATED);
        }
    } else {
        isPendingMessagesExists = false;
    }
    transaction.commit();
    if (isUseJTATransaction) {
        transactionManager.commit();
    }
```

Figure 4-15 InvokerWorker

If the user has set the MercuryUseJTATransaction property at the axis2.xml then InvokerWorker checks for this and start a distributed transaction before invoking the business logic and commit it if there is no exception. At the message receiver it gets a connection from the AtomikosDataSourceBean in order to enlist the transaction with the distributed transaction started at the InvokerWorker. In out and fault handling scenarios are almost same as corresponding persistence case while having above changes to support jta.

Chapter 5

Observations & Results

A WS-RM implementation can be used in different ways with the different types of storages. The reliability and fault tolerance achieved varies according to the type of storage being used. Rest of the chapter describes some of the observations made with the different scenarios mentioned in the earlier chapter.

Any reliable messaging framework downgrades the performance of sending messages. In other words reliability is inversely proportional to the performance. In WS-RM this is mainly because initial sequence creation and acknowledgement messages. Further it takes time to store the message to the persistence storage in the case of persistence and jta scenarios.

In memory model provides the weakest form of reliability. It provides the reliability for network failures but can't survive with the node failures. If the node fails it loses all the messages and sequence state and hence fail to recover.

Persistence model provides better reliability than in memory model. It provides the reliability for network failures. Since it persists sequence state and messages received it can restart RM sequences after a node fail. For this project work we tested this model by sending 20 messages while stopping and starting the client node and server node. Although it can recover sequences we observe some messages has lost. The number of messages at the

APPLICATION_SERVER database was less than 20 for in only case. Further number of reply messages were also less than 20 in APPLICATION_SERVER for in out case.

Persistence storage with JTA support provides the best reliability. First it provides reliability for network failures. Further JTA support provides the reliability for node failures without losing any message. For this project work we tested the JTA support by sending 20 messages while stopping and starting the client node and server node. But there were no message loses either at the APPLICATION_SERVER database or APPLICATION_CLIENT database.

Chapter 6

Conclusion & Future Work

This thesis describes a queued transaction processing based solution using web service reliable messaging in order to guarantee the client side and server side persistence storage updates. It achieves this goal by re engineering the WSO2 mercury with a storage based API. Hence this project presents a storage API based WS-RM implementation which can support distributed transactions. It provides a set of use case scenarios to describe the way to use the new Mercury Implementation and prove its point in reliability. Sample scenarios uses Apache Derby as the database for its persistence storage and Atomikos as the library to provide the JTA support.

The reliability of Mercury is handled by using a state machine model. Although there is a state machine for WS-RM 1.1 specification there is no such a model for WS-RM 1.0 specification. Therefore the state machine model described here which is independent of the implementation can be used for any WS-RM 1.0 specification implementation.

The storage API developed provides explicit support for both in memory and persistence storage implementations. This storage API which is independent of implementation logic can also be used with any WS-RM implementation.

There are some problems with the 2PC recovery with the Apache Derby XA driver and other commonly used opensource database XA drivers. However investigating deeply into these problems and finding out XA drivers that properly support 2PC recovery, goes beyond the scope of this work and we kept it as a possible future work.

Further research can be done to integrate the WS-RM transactions with the application servers. This allows application developers to integrate Enterprise Java Bean Objects transactions with the WS-RM transactions.

This thesis concentrates only on supporting distributed transactions on a WS-RM implementation. But a WS-RM implementation should address a lot of features with different aspects. Hence we kept adding new features such as implementing WS-RM 1.1 support, use single threaded invocations for synchronous communication, WS-RM level error handling and Secure Reliable Messaging as another possible future work.

REFERENCES

[1]] Andrew S. Tanenbaum and Maarten van Steen, *Distributed Systems: Principles and Paradigms.* New jersey U.S.A : Prentice Hall, Inc, 2002

[2] Philip A. Bernstein and Eric Newcomer, Principles of Transaction Processing. San Francisco, CA : Morgan Kaufmann Publishes Inc, 1997. pp 101-116

[3] Don Box et al. "Simple Object Access Protocol (SOAP) 1.1" W3C Note 08 May 2000.

[4] Luis Felipe Cabrera et al. "Web Services Coordination (WS-Coordination)" August 2005

[5] Luis Felipe Cabrera et al. "Web Services Atomic Transaction (WS-AtomicTransaction)" August 2005

[6] Luis Felipe Cabrera et al. "Web Services Business Activity Framework (WS-BusinessActivity)" August 2005

[7] Jens Lechtenb orger "2-PHASE COMMIT PROTOCOL" University of M¨unster, Germany.

[8] Ruslan Bilorusets et al. "Web Services Reliable Messaging Protocol (WS-ReliableMessaging)" February.

[9] Paul Fremantle et al. "Web Services Reliable Messaging (WS-1 ReliableMessaging) Version 1.1" 14 June 2007.

[10] Don Box et al. "Web Services Addressing (WSAddressing)" Augest 2004.

[11] "Distributed Transaction Processing: The XA Specification" X/Open Company Ltd.

[12] "Java Transaction API (JTA)" Sun Microsystems Inc.

[13] "Java Transaction Services (JTS)" Sun Microsystems Inc.

[14] Stefan Tai, Thomas A. Mikalsen, Isabelle Rouvellou "Using Message-oriented Middleware for Reliable Web Services Messaging" IBM T.J. Watson Research Center, Hawthorne, New York, USA

[15] Christoph Liebig and Stefan Tai "Middleware Mediated Transactions" Darmstadt University of Technology, Darmstadt, Germany IBM T.J. Watson Research Center, New York, U.S.A., 2001

[16] Stefan Tai and Isabelle Rouvellou "Strategies for Integrating Messaging and Distributed Object Transactions" IBM T.J. Watson Research Center, New York, USA, 2000

[17] Stefan Tai, Thomas A. Mikalsen, Isabelle Rouvellou, Stanley M. Sutton Jr. "Dependency-Spheres: A Global Transaction Context for Distributed Objects and Messages" IBM T.J. Watson Research Center, New York, U.S.A., 2001

[18] Stefan Tai, Alexander Totok, Thomas Mikalsen, Isabelle Rouvellou "Message Queuing Patterns for Middleware-Mediated Transactions" IBM TJ. Watson Research Center, New York, USA Courant Institute of Mathematical Sciences, New York University, New York, USA, 2002