A CLUSTERING ALGORITHM IN TWO-PHASE COMMIT PROTOCOL FOR OVERCOMING DISTRIBUTED TRANSACTION FAILURE

TERESA KWAMBOKA ABUYA

MASTER OF SCIENCE

(Computer Systems)

JOMO KENYATTA UNIVERSITY OF

AGRICULTURE AND TECHNOLOGY

2015

DECLARATION

This thesis is my original work and has not been presented for a degree in any other University.

Signature……..Date:……

Teresa Kwamboka Abuya

This thesis has been submitted for examination with our approval as University Supervisors

Signature:……..Date:……..

Dr. Richard M. Rimiru

JKUAT, Kenya

Signature:……..Date:……..

Dr. Cheruiyot W.K

JKUAT, Kenya

DEDICATION

This work is dedicated to my mother Lucy K. Abuya who taught me to be hardworking, honest, focused and respective of others. Not forgetting her unending guidance and encouragement while undertaking this thesis. To my late father Francis Abuya for my earlier upbringing. You were the greatest gift that God gave to me.

ACKNOWLEDGEMENT

I am greatly indebted to Dr.Rimiru and Dr.Cheruiyot, my supervisors for their patience, guidance, enthusiastic encouragement and useful critique of this thesis. I am grateful for your words of wisdom, mentorship and emotional support which have made me reach this far. May the good lord always reign in your life. Not forgetting Proff.Sigei, Dr.Stephen Kimani, Dr. Mindila and Sylivester Kiptoo for their invaluable advice in this thesis. Thank you all for being sunshine in my cloudy days, indeed you were my inspiration.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

ABBREVIATIONS AND ACRONYMS

2PC: Two-Phase Commit

3PC: Three-Phase Commit

TM: Transaction Manager

DM: Data Manager

DDS: Distributed Database System

DDBMS: Distributed Database Management System Software

BTM: Bitronix Transaction Manager

API: Application Porgramming Interface

ABSTRACT

The purpose of this research was to simulate two phase commit protocol connectivity based clustering algorithm for overcoming distributed transaction failure. The important issue in transaction management is that if a database was in a consistent state prior to the initiation of a transaction, then the database should return to a consistent state after the transaction is completed. This should be done irrespective of the fact that transactions were successfully executed simultaneously or there were failures during execution. The research objectives were: To analyze the distributed transaction failures in two-phase commit protocol; to simulate a transaction clustering algorithm for overcoming distributed transactions failure in two phase commit protocol; and to compare the performance of the transaction clustering algorithm with the current two phase commit protocol. The experimental research design was adopted as it involved the practical design of the transaction clustering algorithm. This algorithm was simulated in Jcreator, with *mySQL* acting as the backend data manager. The data was collected using Java Integrated Development Environment, which was Jcreator, with Bitronix transaction manager providing the required management of distributed transactions. It was then analyzed using the same software. The results obtained indicated that by using a clustering algorithm, the transaction failures associated with the current Two-Phase Commit can be reduced. This was achieved by eliminating transaction partitioning that is an inherent feature of the current two phase commit protocol. In a partitioned environment, blocking caused by the failure of the coordinator when participants are in uncertain state is a common problem. Instead, all sub-transactions were clustered in a single sub-class and used the principle of inheritance to obtain variables and methods from the main super-class, which was the coordinator. The transaction manager was then employed to coordinate the execution activities of the coordinator. Transaction commit or transaction roll back was then reported by the transaction manager. In so doing, all the transactions in the sub-class either commit in their entirety or fail in their entirety which is in line with the principles of a Two-Phase Commit protocol.

CHAPTER ONE

1.0 INTRODUCTION

1.1 Background Information

Distributed database systems pose different problems when accessing distributed data. An important issue in transaction management is to ensure consistency of database despite failures during execution (Krishna & Masaru, 2011).

A transaction consists of a series of operations performed on a database and the important issue in transaction management is that, if a database was in a consistent state prior to the initiation of a transaction, then the database should return to a consistent state after the transaction is completed. This should be done irrespective of the fact that transactions were successfully executed simultaneously or there were failures during execution. Transactions communicate with Transaction Managers(TM) and TMs communicate with Data Managers (DM) and DMs manage the data being committed (peng-yeng *et.al*, 2006).

A Distributed database system (DDSs) implements a transaction commit protocol to ensure transaction atomicity. Several sites need to update their databases with the same information. One client requests information to be uploaded and a site receive the request and start a procedure where he becomes the coordinator of this request. The other sites become participants of the particular request.

Over several decades a variety of protocols have been proposed by researchers. To achieve their functionality these protocols require exchange of multiple messages in multiple phases between client and coordinator where distributed transaction is done. A failure of one site in committing its part of the transaction could cause the entire system to be inconsistent. Thus some form of control is necessary to ensure that concurrent execution of transactions in a distributed environment does not jeopardize the integrity of the system as well as its data consistency. The performance factor of concurrency

control algorithms depends on systems throughput and transaction response time. Four cost factors influence the performance: local processing, inter-site communication, transaction restarts and transaction blocking (Taibi *et.al*,2009)

Concurrency control uses two types of commit protocols which include the Two Phase Commit (2PC) and Three-Phase Commit (3PC) protocols.2PC protocol is of prime importance to many distributed transaction processing applications used by financial institutions and other applications that fall within the spectrum of enterprise computing. These types of applications are increasingly being used to harness the availability of commodity processing power scattered in many sites of medium to large scale organizations. Only two phases are executed in 2PC .The prepare and commit phase but it has a blocking disadvantage in which either the coordinator or some participating site is blocked.

Amir *et.al* (2010), said that 3PC protocol was introduced as a remedy to the blocking disadvantage of 2PC protocol. It introduced an extra phase called the pre-commit phase which ensured the non-blocking property of this protocol. Although 3PC protocol overcomes blocking problem, it involves an additional round of message transmission to achieve non-blocking property. If 3PC protocol is implemented to eliminate the blocking problem, an extra round of message transmission further reduces the system's performance as compared to 2PC protocol. Especially in DDBS environments, in which frequent site failures and longer message transmission times occur, neither 2PC protocol with blocking problem nor 3PC protocol with performance degradation problem is suitable for the commit processing.

Recently commit processing has attracted a strong attention due to its effect on the performance of the transaction processing. In Tarekhelmy and Fahd(2011),it has been shown that using simulation in distributed commit processing can have more influence than distributed data processing on throughput performance.

Two-Phase Commit(2PC) protocol has a blocking problem when coordinator fails. The blocking problem in 2PC is one of the main issue to solve when designing an efficient distributed system. To solve the blocking problem and show the effectiveness of 2PC, a transaction clustering algorithm is simulated in Two-Phase commit protocol to demonstrate how transactions are committed and how data consistency is maintained in a distributed system with concurrent execution of randomly generated transactions.

Several possible failure cases are identified and created to test its integrity, showing how it responds to different failure scenarios and recovery from failures. The 2PC simulator was achieved using Bitronix transaction manager(BTM),which allows applications to perform distributed transactions, to access and update systems having multiple transaction resources ,databases, message queues and resources accessed from multiple processes or on multiple hosts as participants in a single transaction.

1.2 Fundamentals of transaction management

The concept of a database transaction or atomic transaction has evolved in order to enable both a well understood database system behavior in a faulty environment where crashes can happen any time, and recovery from a crash to a well understood database state. A database transaction is a unit of work, typically encapsulating a number of operations over a database. Each transaction has well defined boundaries in terms of which program/code executions are included in that transaction determined by the transaction's programmer via special transaction commands.

Each transaction has to terminate. The outcome of the termination depends on the success or failure of the transaction. When a transaction starts executing, it may terminate with one of the two possibilities:

i. The transaction aborts if a failure occurred during its execution.

ii. The transaction commits if it was completed successfully.

Part a of figure 1.0 below shows an example of a transaction that aborts during process 2 (p2).On the other hand part b of figure 1.0 shows an example of a transaction that commits since all of its processes are successfully completed.

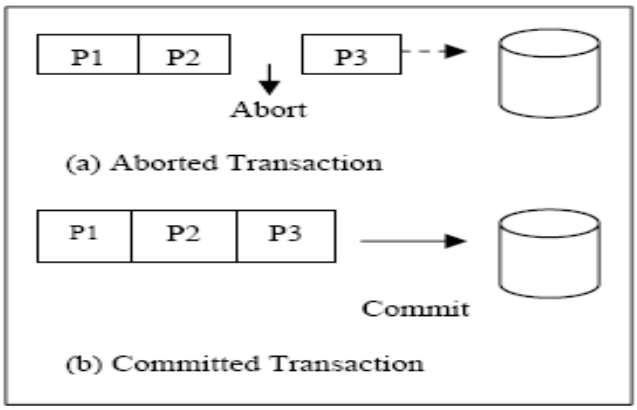

Figure 1.0 Abort and commit transactions

Every database transaction obeys the following rules in a database system: (Coulouris *et.al*, 2007). Atomicity: Atomicity guarantees that many operations are bundled together and appear as one contiguous unit of work, operating under an all-or-nothing paradigm. Either all of the data updates are executed or nothing happens if an error occurs at any time. In other words, in the event of failure in any part of the transaction, all data will remain in its former state as if the transaction was never attempted. In transactional terminology, this is referred to as rolling back the transaction.

Consistency: Every transaction must leave the database in a consistent state, i.e., maintain the predetermined integrity rules of the database like constraints upon and among the database's objects. A transaction must transform a database from one consistent state to another consistent state. Since a database can be normally changed only by transactions; all the database's states are consistent. An aborted transaction does not change the database state it has started from, as if it never existed. For example dirty data, is data that has been modified by a transaction that has not yet committed. Thus the function of concurrency control is to disallow transactions from reading or updating dirty data.

Isolation: This property in distributed systems protects concurrently executing transactions from seeing each other’s incomplete results. Isolation allows multiple transactions to read or modify data without knowing about each other because each

transaction is isolated from the others. Each transaction should see a consistent database at all times. Providing isolation is the main goal of concurrency control.

Durability: This property ensures that once a transaction commits, its results are permanent and cannot be erased from the database. This means that whatever happens after the COMMIT of a transaction, whether it is a system crash or aborts of other transactions, the results already committed are not modified or undone.

1.3 Atomic commit protocols

Atomic commit protocols are used in distributed systems when several sites need to update their databases with the same information. One client requests information to be uploaded and a site receive the request and start a procedure where he becomes the coordinator of this request. The other sites in the system will then become participants of the particular request. The atomicity property of the protocol means that the transaction must be performed at all sites or not at all; this is achieved by letting all participants vote YES or NO to the particular transaction depending if they can commit it or not. Only when all sites are ready to commit, the coordinator sends a GLOBAL_COMMIT to the participants as confirmation that they can commit the transaction. A site can be both coordinator and participant at the same time but for different transactions. If a coordinator site crashes and a participants waits for a final answer from the coordinator, if he should commit the transaction or not, the participants is blocked as long as the coordinator is down(Groote *et.al*, 2008). This is the problem with the 2PC algorithm, while 3PC have more communication between the coordinator and the participants, and can therefore avoid this problem. Though, all communication takes valuable time in a distributed system where the sites can be far away from each other, a 2PC-protocol is to prefer. This study analyses 2-Phase Commit and 3-Phase Commit protocols. It looks at the types of failures introduced by a distributed system (Kumar et.al, 2011).

1.4 Problem statement

The world is moving towards a trend where tasks are performed in a distributed manner. Goebel(2011), noted that the main aim of distributed transaction management is to achieve atomicity across all sites and reduce transaction failure. Distributed database systems like airline reservation systems, banking applications, credit card systems widely use these protocols for their transactions over the network. According to Helmy and Alotaibi(2011), Two-Phase Commit and Three-Phase Commit protocols have been researched on in as far as concurrency control is concerned. 2PC protocol has a problem of blocking caused by the failure of the coordinator when participants are in uncertain state. Similarly 3PC was employed to deal with blocking since it is a non-blocking algorithm but it is complicated to implement, has more communication overheads and maintaining inconsistency towards network partitioning problems (Tabassum *et.al*, 2011). Therefore, to improve the concurrency control and blocking problems in 2PC, a transaction connectivity based clustering algorithm is proposed which reduces transaction blocking considerably, while at the same time enhancing concurrency control thus increasing the efficiency of transaction processing.

1.5 Objectives of the study

1.5.1 General Objective

The general objective of this study was to simulate a Two-phase commit protocol connectivity based clustering algorithm for overcoming distributed transaction failure.

1.5.2 Specific Objectives

i) To analyze distributed transaction failures in Two-phase commit protocol.

ii) To simulate a connectivity-based clustering algorithm for overcoming distributed transactions failure in two phase commit protocol.

iii) To compare the performance of connectivity based clustering algorithm with the current two phase commit protocol.

1.6 Research questions

1. How do distributed transaction failures occur in two-phase commit protocol?
2. What will be the effect of connectivity based clustering algorithm in overcoming failures in distributed transactions in two phase commit protocol?
3. How does the connectivity based clustering algorithm perform in comparison with the current two phase commit protocol?

1.7 Justification

A distributed database combines two different technologies used for data processing: database systems and computer networks. Query processing from different sites is more complex and difficult in distributed environment. This study attempts to achieve good transaction and concurrency control performance of the two phase commit protocol. Utilization of the connectivity based transaction clustering algorithm on a 2PC protocol minimizes the response time, improves concurrency control in 2PC protocols, reduce transaction inconsistency, eliminate transaction blocking and reduce high communication overheads. This will go along way in maximizing transaction throughput.

1.8 Scope of the study

The study focused on connectivity based transaction management in a concurrent distributed systems environment. In connectivity based clustering algorithms, the concept of object oriented objects is used. In this perspective, objects are more related to nearby objects than to objects farther away. The intention of this approach was to determine how commit protocols are used in executing transaction requests and address challenges faced in transaction execution to achieve atomicity property.

1.9 Limitations of the study

This research thesis was limited to addressing distributed transaction failures experienced in Two Phase Commit protocols during transaction processing.

CHAPTER TWO

2.0 LITERATURE REVIEW

2.1 Introduction

This literature looks at different protocol frameworks developed to address existing problems in distributed processing.ACID(Atomicity,Consistency,Isolation,Durability) properties of transactions have been visited and various commit protocols have been discussed like, Two Phase Commit and Three Phase Commit protocols with an aim of identifying their shortcomings. The efficiency of a commit protocol is associated with the number of communication steps, the number of log writes and its execution time, and the coordinator and each participant. The Blocking or non- blocking nature and difference in recovery procedures are other important factors that have a vital impact on the overall commit protocol performance (Reddy & Kitsuregawa, 2006).

2.2 Overview of transactions

A transaction is defined to provide the properties of atomicity, consistency, integrity and durability (ACID) for any operation it performs. In order to ensure the atomicity of distributed transactions, an atomic commit protocol needs to be followed by all sites participating in a transaction execution to agree on the final outcome, that is, commit or abort. A variety of commit protocols have been proposed that either enhance the performance of the classical two-phase commit protocol during normal processing or reduce the cost of recovery processing after a failure.

In this study we survey Two-Phase Commit(2PC) and Three-Phase Commit(3PC) protocols and optimizations providing an insight in the performance trade-off between normal and recovery processing. We analyze the performance of a representative set of commit protocols analytically using simulation(Kotla *et.al,* 2010).

Transactions are powerful abstractions that facilitate the structuring of database systems and in distributed systems in general in a reliable manner. Each transaction represents a task or a logical function that involves access to a shared database and assumes as it executes as if no other transactions were executing concurrently and as if there were no program and system failures. In this way programmers are relieved from dealing with the complexity of concurrent programming and failures, and can focus on designing the applications and developing correctly the individual transactions of the applications.

A transaction provides reliability guarantee by implementing a state transformation with four important properties, commonly known as ACID properties.

Atomicity ensures that either all or none of the transaction's operations are performed. Thus, all the operations of a transaction are treated as a single, indivisible, atomic unit. Similarly consistency requires that a transaction maintains the integrity constraints on the database. Isolation on the other hand demands that a transaction executes without any interference from other concurrent transactions. Finally durability ensures that all the changes made by a successfully terminated transaction become permanent in the database, surviving any subsequent failure.

The ACID properties are usually ensured by combining two different sets of algorithms. The first set, referred to as concurrency control protocols, ensures the isolation property, whereas the second one, referred to as recovery protocols, ensures atomicity and durability properties. Commonly consistency is satisfied by designing transactions such that each transaction preserves the consistency of the database at its boundaries and is enforced by specifying integrity constraints on a database using triggers and alerters(Parul *et.al*,2011)

In a distributed database system (DDBS) in which the data items are stored at multiple sites interconnected via a communication network, transactions are executed in a distributed fashion at different sites based on the location of the data that they require to access. Since sites and communication links can fail independently, the atomicity

property of a distributed transaction cannot be guaranteed without taking additional measures besides concurrency control and recovery protocols. Specifically, for a distributed transaction that executes across multiple sites, the sites need to agree about when and how the transaction should terminate. That is, all the sites participating in a transaction execution need to (1) eventually reach an agreement; and(2) all agree to either commit the transaction, making all its effects persistent, or abort the transaction, obliterating all its as if the transaction had never executed. A protocol that achieves this kind of agreement is called an atomic commit protocol (ACP).

2.3 Theoretical Analysis of distributed transaction failures in Two-Phase Commit Protocol (2PC)

A commit protocol is an algorithm to ensure atomicity in a distributed transaction with the help of synchronized locking. According to Tannenbaum (2007), Atomic commit protocols are used in distributed systems when several sites need to update their databases with the same information. Several protocols exist that have been used to address atomicity in different protocol platforms. The following are some of the protocols discussed:

2.3.1 Two-Phase Commit (2PC) Protocol scenario

The atomic Two-Phase Commit Protocol (2PC) is a typical distributed commit algorithm used in computer networks and distributed database systems. It has two phases i.e the prepare and commit phase. It is used when a simultaneous data update should be applied within a distributed database. In this protocol, one node acts as the coordinator, which is also called master and all the other nodes in the network are called participants or slaves. The prepare messages from a participant to a coordinator are *YES* or *No* depending on the decision at the participant whether to vote yes or no to the requested transaction. The commit messages from a coordinator to the participant are *GLOBAL_COMMIT* or *GLOBAL_ABORT* depending if all participants have voted yes or not. All decisions at each site are logged in their respective write-ahead-log along with the transaction. The

write-ahead-log must be in a stable storage to ensure that data is not lost during a site failure. In its first phase, all these participants agree or disagree with the coordinator to commit, i.e., vote yes's or no's and in 2nd phase they complete the transaction simultaneously by getting the commit or the abort signal from the coordinator.

Global commit or abort means all participants must commit or abort, even if there is failure or timeout at any one of the nodes. Timeout means the failure of the other site. The coordinator plays the central role and flags either global commit or global abort. The former is only shown if all the participants vote to commit and the latter is shown if at least one of the participants votes to abort or the coordinator decides to abort the current transaction. In case there is no failure at any site, the protocol is correct but it is highly desirable to consider the functionality in the presence of failure of any site at any state(Cowling et.al, 2010).

Figure 2.0 below shows a Two-Phase Commit (2PC) protocol scenario.

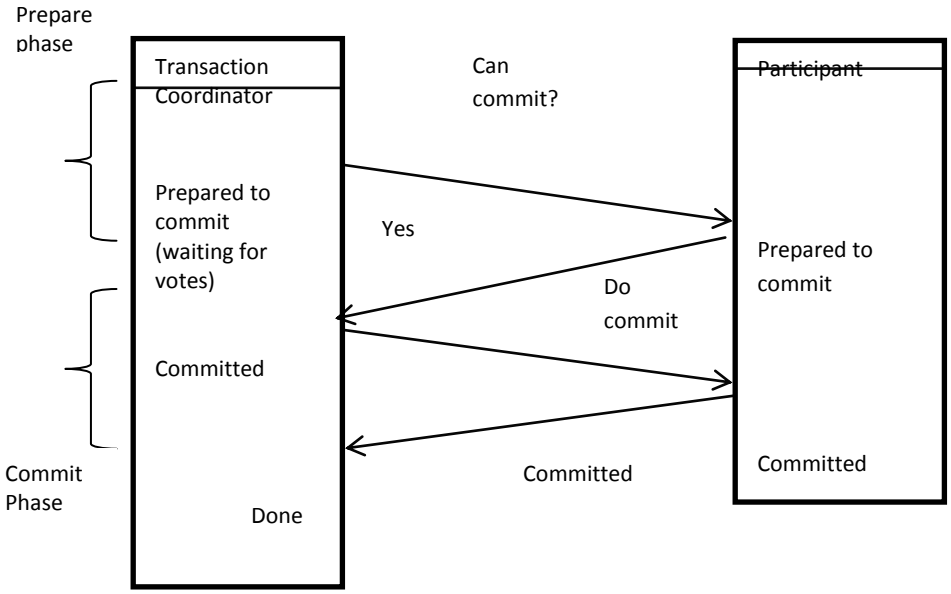

Figure 2.0 Two Phase Commit Protocol architecture (Jumna *et.al,* 2012)

Consider the scenario that 2PC protocol does not have any failures and the operations are as follows:

a) Prepare phase

Coordinator: Initially the coordinator will broadcast the Begin_commit request message to all participants and enters into wait state.

Participant: When the participant receive the request message, If the participant want to commit the transaction means it respond with the Vote_commit message(Yes) to the coordinator and enters into ready state. Otherwise, the participant responds with the Vote_abort message (No) to the coordinator.

Coordinator: When the coordinator receives the reply from participant it starts 2nd phase.

b) Commit phase

Coordinator: If the participants reply with Vote_commit message(Yes), the coordinator decided to commit the transaction or abort the transaction and it will inform the participant about the outcome of the transaction.

Participant: The Participant follows the coordinator's command and it will acknowledge the coordinator.

However 2PC protocol having less communication overhead and less expensive, it has a main drawback that is blocking transaction problem(Jamuna*et.al*, 2012).

2.3.2 Two-Phase Commit protocol variants

There are three main variants of 2PC which deal with how to handle recovery and vary on how recovery data is logged.They include:-

a)Presumed Nothing (PrN)

b)Presumed Abort (PrA)

c)Presumed Commit (PrC)

a) Presumed Nothing(PrN)

This is the basic version of Two-Phase Commit andin this the coordinator requires very explicit information which forms the word “Presumed nothing” or protocol.

The Protocol Messages for PrN

To commit a distributed transaction, PrN requires two messages from coordinator to cohort and two messages from cohort to coordinator, or four messages in all. The protocol has the following steps:

i. The coordinator sends PREPARE messages to all cohorts to notify them that the transaction is to be terminated.

ii. Each cohort then sends a vote message either a COMMIT-VOTE or an ABORT-VOTE on the outcome of the transaction. A cohort responding with a COMMIT-VOTE is now prepared.

iii. The coordinator commits the transaction if all cohorts send COMMIT-VOTES. If any cohort sends an ABORT-VOTE or the coordinator times out waiting for a vote, the coordinator aborts the transaction. The coordinator sends outcome messages i.e. COMMIT or ABORT to all cohorts.

iv. The cohort terminates the transaction according to its outcome, either committed or aborted, and then ACKs the outcome message(Samaras *et.al*, 2003)

Cohort Activity

A cohort must log enough information stably so that it can tolerate failures both before the commit protocol begins and during the commit protocol. If a cohort fails, it’s necessary to abort every transaction that has had any activity there and is not yet prepared there. Otherwise updates might be lost, or serializability might be compromised because read locks are released prematurely as a result of the failure.

Hence the cohort must vote to abort a transaction if the cohort has failed since the first time it saw any activity for the transaction. Two ways to ensure this which do not require any logging are given below.

i. The client marks the first action of a transaction that it sends to each cohort. The cohort records a transaction as active when it sees an action marked as first, and votes to abort a transaction unless it's recorded as active.

ii. The cohort counts the number of actions it has seen for each transaction, and the client counts the number of actions it has sent to each cohort. The client passes all the counts to the coordinator with the commit request and the coordinator passes each count on to the proper cohort. The cohort votes to abort if its count is different.

Before responding with a COMMIT-VOTE, a cohort must stably record that it is prepared. This makes it possible for it to commit the transaction even if it is later interrupted by a crash. If a prepared cohort does not receive a transaction outcome message promptly, or crashes without remembering the outcome, the cohort asks the coordinator for the outcome. It keeps on asking until it gets an answer. This is the blocking aspect of 2PC(Grey & Reuter, 2001)

b) Presumed Abort

In the absence of information about a transaction in its protocol database, a presumed abort (PrA) coordinator presumes the transaction has aborted. This abort presumption was already made occasionally by PrN. PrA makes it systematically to further reduce the costs of messages and logging. Once a transaction has aborted, its entry is deleted since a missing entry denotes the same outcome. No information need be logged about such transactions because their protocol database entries need not be recovered. We must guarantee that the protocol database always contains entries for committed transactions which have not yet completed all phases of 2PC.

These entries must be recoverable across coordinator crashes. This means that as in PrN, the coordinator must make transaction commit stable before sending a COMMIT message, by forcing this outcome to its log. PrA deletes the protocol database entries for committed transactions when 2PC completes in order to limit the size of the database, just as PrN does. And the same garbage collection strategies are also possible.

A coordinator need not make a transaction's entry stable before its commit because an earlier crash aborts the transaction, and that is the presumed outcome in the absence of information. Only a commit outcome needs to be logged with a forced write. Since there is no entry in the protocol database for an aborted transaction, there is no entry in need of deletion, and hence no need for an ACK of the ABORT outcome message.

In summary, PrA aborts a transaction more cheaply than PrN, and it commits one in exactly the same way.

c) Presumed Commit

For presumed commit (PrC), the coordinator explicitly documents which transactions have aborted. While this has some apparent symmetry with PrA, which explicitly documents committed transactions, in fact there is a fundamental difference. With PrA, we can be very lazy about making the existence of a transaction stable in the log. If there is a failure first, we presume it has aborted. But PrC needs a stable record of every transaction that has started to prepare because missing transactions are presumed to have committed, and a commit presumption is wrong for a transaction that fails early. Traditionally this has meant that at the time 2PC is initiated and a transaction is entered into the protocol database, the coordinator forces a transaction initiation record to the log to make its database entry stable. This entry can then be recovered after a coordinator crash, so that an uncommitted transaction is aborted rather than presumed to have committed. With PrC, a transaction's entry is removed

from the protocol database when it commits, because missing entries are presumed to have committed. If Cohorts subsequently inquire, they are told the transaction committed by presumption. Thus, PrC avoids ACK messages for committed transactions, which is the common case and hence a significant saving much more important than avoiding acknowledgements(ACKs)s for aborted transactions(Lomet & Salzberg, 2003).

We must ensure that a committed transaction's entry is not re-inserted into the protocol database when the coordinator recovers from a crash. If this happened, we might think the transaction should be aborted. Hence, like PrN and PrA, PrC forces commit information to the log before sending the COMMIT message. Logically, this log write erases the initiation log record, since lack of information implies commit. However, given the nature of logs, it is easier to simply document the commit by forcing a commit record to the log tail. The commit log record tells us not to include the transaction in the protocol database of aborted transactions.

With PrC, both the protocol database entry and the initiation log record list all cohorts from which ACKs are expected if the transaction aborts. When all the ACKs have arrived, the entry can be garbage collected from the protocol database. Like PrN, PrC writes a non-forced end record to the log at this point to keep the transaction from being re-entered into the protocol database. No separate abort record is needed.

In summary, PrC commits a transaction with two forced log writes, the initiation record and the commit record. In addition, it sends two messages to each cohort, PREPARE and COMMIT. In response, each cohort forces a prepare log record and writes a commit log record. The commit record need not be forced because a prepare record without a commit record causes the cohort to inquire about the outcome. The coordinator, not finding the transaction in its protocol database, will respond with a COMMIT message.

The coordinator removes read-only cohorts from the list of cohorts that should receive the transaction outcome message. If every cohort sends a READ- ONLY-VOTE, then the coordinator sends no out- come message. In addition, it no longer matters whether the transaction is considered committed or aborted. Hence the coordinator can choose whichever outcome permits the least logging. This is not a good transaction management mechanism (Manikandan *et.al*, 2012)

Blocking problem in 2PC

The Blocking problem is described with the given circumstances that, if the coordinator fails to operate and at the same time some participant has confirmed itself to commit state. The participants keep locks on resources until they receive the next message from the coordinator after its recovery. For instance consider a situation that a participant has sent VOTE-COMMIT message to the coordinator and has not received either GLOBAL-COMMIT or GLOBAL-ABORT message due to the coordinator's failure. In this case, all such participants are blocked until the recovery of the coordinator to get the termination decision. The blocked transactions continue to keep all the resources until they obtain the final decision from the coordinator after its recovery (Schapiro & Milistein, 2012).

2.4 Types of failures introduced in distributed transactions

From the literature above, 2PC and 3PC commit protocols have been discussed and it has been established that circumstances under which distributed transactions are committed or undone under 2PC include:

- When application instructs the transaction to rollback, then the transaction will be undone.
- When process failure occurs before all participant votes, then transaction will be undone.
- If any participant votes no, then transaction will be roll backed.
- If all participants vote yes, transaction will be committed.

- If Process failure occurs after all participants have voted and the transaction coordinator has received all voters as yes, then transaction will be committed but is unresolved.

It is noted that the 2PC goes to a blocking state by the failure of the coordinator when the participants are in uncertain state. The participants keep locks on resources until they receive the next message from the coordinator after its recovery (Manikandan *et.al*, 2012). Thus, all such participants are blocked until the recovery of the coordinator get a termination decision. Although 3PC protocol eliminates the blocking problem, it involves an extra overhead of one more cycle and in turn increases time taken for the transaction to complete (Singh *et.al*, 2011).

Three types of failures are introduced in distributed transaction environment as proposed by Byun and Moon(2012).They include:-

2.4.1 Site Failure

A failure of any type is normally detected by the absence of an expected message. Site failures are usually due to software or hardware failures. These failures result in the loss of the main memory contents. In distributed database, site failures are of two types:

i. *Total Failure* where all the sites of a distributed system fail

ii. *Partial Failure* where only some of the sites of a distributed system fail.

Site failures are modeled by a failure transition, which is a special kind of local state transition. Such a transition occurs at the failed site the instant that it fails. The resulting local state is the state initially occupied by the failed site upon recovering. An underlying assumption is that a site can detect when it has failed.

2.4.2 Network partitioning failure

A network failure results in at least two sites that cannot communicate with each other. Research done on network partitioning proposed to model such a partition in two ways. The first model is a pessimistic model where all messages are lost at the time partitioning occurs. The second model is an optimistic model where no messages are lost at the time partitioning occurs; instead, undeliverable messages are returned to the

sender. While the pessimistic model is more realistic, the optimistic model is theoretically interesting since it yields upper bounds on the achievable resiliency.

A simple partitioning occurs when the sites are partitioned into exactly two sets with no communication possible between the sets. A multiple partitioning occurs when the sites are partitioned into *k* sets. A multiple partitioning can be viewed as simultaneous occurrences of two or more simple partitioning. A protocol is resilient to a network partitioning only if it is non-blocking, that is, the protocol must ensure that each isolated group of sites can reach a commit decision consistent with the remaining groups. Unless otherwise stated, we will assume that partitions are caused by link failures rather than site failures (Taranum *et.al,* 2011)

2.4.3 Coordinator Failure

The 2PC protocol is the simplest and the best known protocol which serves as an object to ensure the atomic commitment of a distributed transaction. It is a centralized control mechanism based on the coordinator, which coordinates the actions of the others called participants. A coordinator sends transaction request to participants and waits for their replies in the first phase. After receiving all replies, the coordinator sends a final decision to participants in the second phase.

The atomicity property of the protocol means that the transaction must be performed at all sites or not at all; this is achieved by letting all participants vote YES or NO to the particular transaction depending if they can commit it or not. Only when all sites are ready to commit, the coordinator sends a GLOBAL-COMMIT to the participants as confirmation that they can commit the transaction. A site can be both coordinator and participant at the same time but for different transactions. If a coordinator site crashes and participants waits for a final answer from the coordinator, if he should commit the transaction or not, the participants are blocked as long as the coordinator is down. This is the problem with 2PC algorithm which reduces high degree of data availability.

2.5 Failure handling principles in distributed systems

When it comes to failures most of it fall into one of the two buckets: Hardware or software failure.

Hardware failure: It used to be more common, but with all of the recent innovations in hardware design and manufacturing they tend to be fewer and far between with , most of these physical failures tending to be network or drive-related.

Software failure: It comes in many more varieties and software bugs in distributed systems can be difficult to replicate and consequently fix.There are two important techniques of handling failure which includes:-

2.5.1 Fault tolerance

An important part of service based architecture is to set up each service to be fault tolerant, such that in the event of one of its dependencies are unavailable or return an error it is able to handle those cases and degrade gracefully. There are many methods of achieving fault tolerance in a distributed system and these includes:

Standbys: This is exactly that a redundant set of functionality or data waiting on standby that maybe swapped to replace another failing instance. Replication can be utilized to maintain real time copies of master database so that data may be replaced without loss or disruption.

Feature flags: It's used to enable or disable functionality in a production system. In the event of failure of a particular system features that depend on that system can be turned off and be made unavailable until that system comes back online.

Asynchrony: Its one of the most important design considerations in any distributed applications. Each service or functional piece of a system communicates with each of its external dependencies asynchronously so that slow or unavailable services do not directly impact the primary functioning of the application. This also implies typically that operations aren't tightly coupled requiring the success of one operation for another to succeed like a transaction and don't require services to be available to handle requests(Dollmore *et.al,* 2007)

2.5.2 Monitoring

Extensive monitoring and logging is essential to nay complex distributed system. Having many services each with a different purpose yet still interacting with one another can lead to highly unusual bugs when they occur. It can be hard to tell where the problem lies and where the issue need to be resolved. One of the best ways to mitigate this confusion and help diagnose problems quickly is to be sure that all system interfaces and APIs are monitored. However, monitoring in large scale web systems can be challenging. Separation and services adds complexity.

There many different points to control and they don't necessarily operate in sync with one another and communication between these systems can be delayed and complicated through mechanisms like retries which only compounds the problem of tracing sequential events.

A big challenge is that too much monitoring or logging can cause delays, take up space and potentially interfere with normal operations.

2.6 Three-Phase Commit(3PC) scenario

A 3PC is a non-blocking protocol which eliminates the blocking problem faced by 2PC protocol. Three Phase Commit protocol operations is similar with the two phase commit protocol, only difference is it has extra phase called Pre_commit phase where it takes the preliminary decision. The Three Phase Commit protocol (3PC) performs the operation in three phases are Prepare phase, Pre-commit phase, Commit/Abort phase. Among the three phases Pre-Commit phase eliminates the blocking problem but it comes with an extra cost of message transfers.

It's represented in the diagram below;

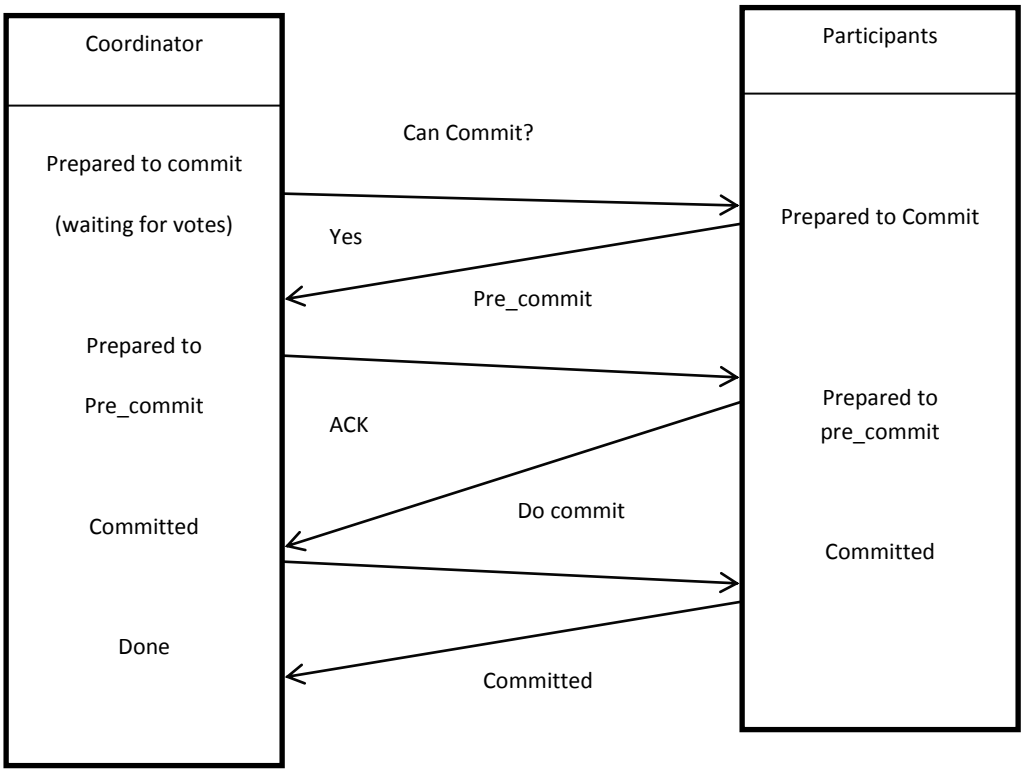

Figure 2.1 Three-Phase Commit Protocol architecture(Jamuna *et.al*, 2012)

a) Prepare phase

Coordinator: Initially the coordinator will broadcast the Begin_commit request message to all participants and enters into wait state.

Participant: When the participant receive the request message, If the participant want to commit the transaction means it respond with the Vote_commit message(Yes) to the coordinator and enters into ready state. Otherwise, the participant responds with the Vote_abort message(No) to the coordinator(Jamuna *et al*, 2012).

Coordinator: When the coordinator receives the reply from participant it starts 2nd phase.

b) Pre-Commit phase

Coordinator: When the coordinator receives Vote_commit message within the time from the participant, the coordinator broadcast the Pre-Commit message to all

participants .At this phase preliminary decision can be made and it moves to prepared state.

Participant: When the participant accepts the Pre_commit message it will send acknowledge message the coordinator.

Coordinator: When the Coordinator receive ACK message from participant it starts 3rd phase.

c) Commit/Abort phase

Coordinator: The coordinator decided to commit the transaction or abort the transaction and it will inform the participant about the outcome of the transaction.

Problems with 3PC

Three-Phase Commit Protocol is problematic only when there are multiple site failures. For example, let's consider a case where the coordinator is in pre-commit state and fails just after sending a commit message and the slave also fails just before or after receiving this message as So by its failure, the slave moves to the aborted state but according to the protocol specifications, the coordinator goes to the committed state, either it fails or receives acknowledgement. Hence, the coordinator moves to the committed state without receiving acknowledgement and the failed slave moves to the aborted state without sending the acknowledgement. In this way, coordinator and participant show different final states due to their failures. Although 3PC protocol eliminates the blocking problem, it involves an extra overhead of one more cycle and in turn increases time taken for the transaction to complete (Singh *et.al*, 2011).Because of high communication overhead 3PC has not been implemented so far.

2.7 Atomic commitment

Atomic commitment in 2PC is an essential demand for the algorithm. It means that every transaction must be handled the same way at all sites, either commit the transaction or abort it. According to Syam (2005),to achieve atomic commitment we need to fulfill the following conditions:

C1.Every participant and the coordinator must reach the same decision.

C2. A GLOBAL_COMMIT will be reached only if all participants and the coordinator have voted commit.

C3.When every participant and the coordinator votes commit and there are no failures the result must be GLOBAL_COMMIT.

C4.A reached decision by a participant or coordinator is not reversible.

C5.If tolerated failures occur and get repaired in a reasonable short time, then all participants and coordinator finally will reach a decision.

2.8 Summary of commit protocols

From the analysis of commit protocols the following points have been noted: That the 2PC has a blocking problem when participants are in uncertain state, and that 3PC though a non-blocking protocol has more communication overheads which hinders performance of distributed transactions

In 2PC protocol, consider a situation that a participant has sent *VOTE_COMMIT* messages to the coordinator and has not received either *GLOBAL_COMMIT* or *GLOBAL_ABORT* message due to coordinator's failure. In this case all such participants are blocked until the recovery of the coordinator to get the termination decision.

Typically one coordinator node has all the information necessary to determine whether a transaction should commit or rollback. Therefore if the coordinator node fails during a distributed transaction, all the participants in the transaction must wait for the coordinator to recover before completing the transaction. Thus significant delays may be caused when the coordinator fails.

To minimize the delay caused by the failed coordinator, this study simulated a 2PC protocol clustering algorithm to optimize distributed transaction failure in improving reliability of distributed transactions.

Transaction processing must ensure transaction atomicity and consistency for transactions that involve databases. Transactions often involve multiple steps all of which must be completed before a database commit can be executed.

Different transaction failure scenarios have been visited. Site failure happens due to software or hardware failures which result in the loss of the main memory contents and in partial or total failure. Network partitioning results in at least two sites that cannot communicate with each other. Coordinator failure in 2PC brings about blocking problem and 3PC doesn't solve it either as it degrades system performance. To address these problems this study used a transaction clustering algorithm on 2PC protocol to minimize transaction failure by eliminating blocking problem in 2PC and increase system performance to achieve high transaction throughput(Fahd *et.al*,2011)

CHAPTER THREE

3.0 METHODOLOGY

3.1 Introduction

This chapter describes steps adopted to design and simulate a connectivity based transaction clustering algorithm for overcoming distributed transaction failure in two-phase commit protocol. This study used an experimental research design that involved the practical design of the transaction clustering algorithm using *Jcreator IDE.* This algorithm was simulated with *mySQL* acting as the backend data manager. The data was collected using Bitronix transaction manager which provided the required management feedback information on distributed transactions. This involved the designation of transaction manager, coordinator, sub-transactions, data managers and participants. This is outlined in the sub-sections below.

3.2 Analysis of distributed transaction failures in Two-Phase Commit protocol

3.2.1 General Requirements of a commit protocol

The general requirements of a commit protocol are presented below:-

R1.The coordinator aborts the transaction if at-least one participant votes to abort.

R2. The coordinator commits a transaction only if all the participants vote to commit.

R3.All non-faulty participants including the coordinator should eventually decide to abort or commit.

R4.If any one of the participants including the coordinator decides to abort or commit then no other participant will decide to commit or abort.

3.3 Tools and Specifications

In order that the researcher shows the coordinator and site failure in two-phase commit protocol, the following tools and specifications were adopted:-

i) *Britonix Transaction Manager (BTM)* - Is a simple but complete implementation of the Java Transaction API (JTA) 1.1 API(application programming interface). It is a fully working XA transaction manager that provides all services required by the JTA API while trying to keep the code as simple as possible for easier understanding of the XA semantics.

ii) *MySQL database Management System* - to act as the backend data resource manager. This should be *MySQL* 5.1 or higher version.

iii) Java Development environment – to provide the virtual machine environment for simulation purpose.

3.3.1 Bitronix Transaction Manager(BTM)

The BTM is a simple but complete implementation of Java transaction applications whose goal is to provide a fully working transaction manager that provides all services required by the Java applications while trying to keep the code as simple as possible for easier understanding of semantics. BTM is such important in transaction management in that it has useful error reporting and logging methods which make it easier to know when an error occurs. BTM configuration settings are stored in a Configuration object. It can be obtained by calling *TransactionManagerServices.getConfiguration().*

BTM is a perfect choice for a project using transaction capabilities by using Java Transfer manager (JTM) facade. It is possible to integrate BTM in web containers like Tomcat or Jetty and get raw access to a JTA implementation.. BTM has proved to be stable and mature enough to be used in production. Currently BTM is very stable and usable. JDBC resources are working pretty well and recovery of crash works fine.

3.3.2 Bitronix JTA Transaction Manager With *MySQL*

The Java Transaction API (JTA) allows applications to performs distributed transactions, to access and update systems having multiple transaction resources:

databases, message queues, custom resource, or resources accessed from multiple processes or on multiple hosts as participants in a single transaction.

3.4 Simulation of 2PC Coordinator Failure

To demonstrate the fact that the Two –phase is blocking, the researcher used Bitronix transaction manager, *mySQL* server and *Jcreator IDE* to simulate a two-phase commit protocol failure. Two practical implementations of coordinator failures were carried out. The first one demonstrated coordinator failure for distributed transactions on a single database while the second one demonstrated coordinator failure for distributed transactions on distributed data resources. In both cases:

i) The *mySQL* database acted as *resource manager*.

ii) The JDBC driver, in this case, *mySQL-connector-java-5.1.10-bin.jar*, acted as *Resource Adapter.*

iii) The *main Java* classes in the projects, acted as *Coordinators. T*wo main classes, were used, namely, the *TwoPCCoordinatorFailureClass* and *TwoPCCoordinatorFailureClass1*. The first main class,*TwoPCCoordinatorFailureClass* was to show the coordinator failure for distributed transactions involving one database while the second class, *TwoPCCoordinatorFailureClass1* was to show coordinator failure for distributed transactions on distributed databases.

iv) *Bitronix Transaction Manager (BTM)* was used *as a Transaction Manager.* Its function was to receive messages from the coordinator and participant and forwards the messages to the corresponding participants and coordinators.

v) The transaction classes, keep *transaction ID*. For example, in Appendix I, the code *"bitronix.tm.BitronixTransactionManager.<init>(BitronixTransactionManager.java:64)" has transaction ID of 64.*

vi) The *transaction sub-classes* send a request for the transaction to the transaction manager-through message calling. For example, in Appendix II, the code

"*btm.commit();",* is a call made to the Bitronix transaction manager to commit a transaction.

vii) A *transaction branch*- is associated with a request to each resource manager involved in the distributed transaction. Each transaction branch must be committed or rolled back by the local resource manager. For example, in Appendix II, the code,

"*catch (Exception ex) {*

ex.printStackTrace();

try {

btm.rollback();

} catch (Exception e) {

e.printStackTrace();

}"

Is a transaction branch that results when the transaction class cannot commit a transaction.

The relationships among these entities are shown Figure 3.1 below. The transaction manager was responsible for making the final decision either to *commit* or *rollback* any distributed transaction. A commit decision should have lead to a successful transaction; rollback leaves the data in the database unaltered. JTA specified standard Java interfaces between the transaction manager and the other components in a distributed transaction: the application, the application server, and the resource managers.

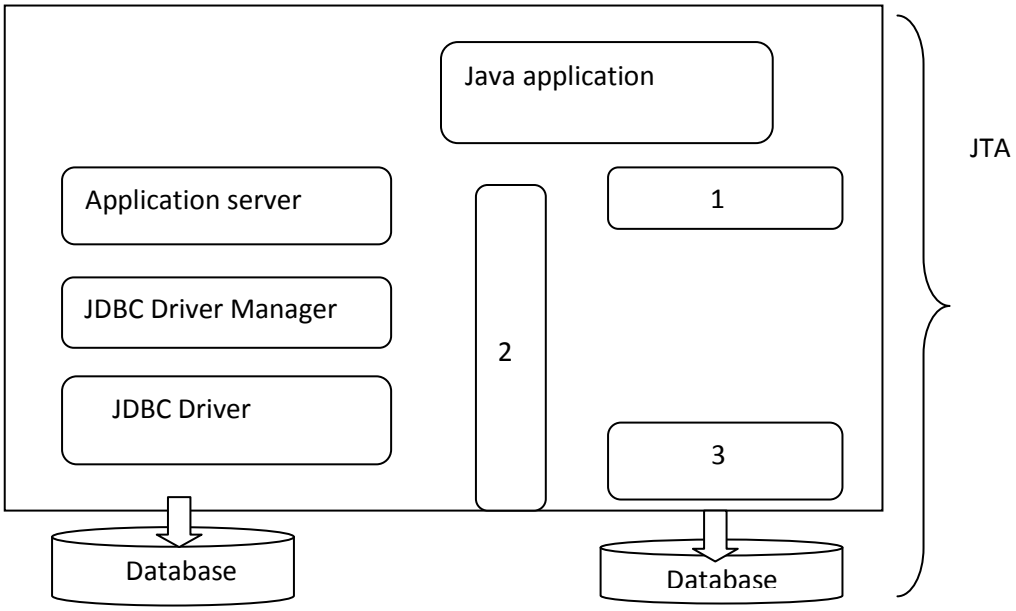

Figure 3.0 Relationship among Distributed System Entities

The numbered boxes, 1, 2 and 3 around the transaction manager correspond to the three interface portions of JTA. The box number 1 is the *userTransaction*, which is an interface that provides the application the ability to control transaction boundaries programmatically. The second (2) is the *transaction manager*, which is an interface that allows the application server to control transaction boundaries on behalf of the application being managed. Lastly, the *XAResource* is box number 3, and is a Java mapping of the industry standard XA (extended Architecture). XA is used for communication with the transactional resources. The two databases were created in *MySQL* and were named *KisiiBranch* and *NairobiBranch*.

3.4.1 Procedure

1. Two databases were created in *MySQL* server. These were given names *KisiiBranch* and *NairobiBranch*.
2. Two tables were created, one in each of these databases, with the name *bankcustomer.*

3. Table *bankcustomer* had five columns, namely *CustomerID, CustomerName ,Address, City and AccountBalance*. Table 3.0 below shows the structure of these tables.

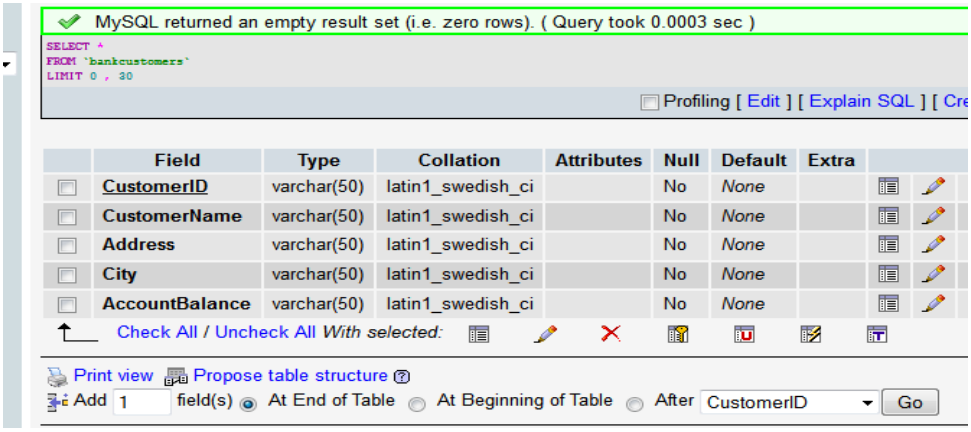

Table 3.0: Structure of the database Tables, *Bankcustomers*

The two phase commit connectivity based clustering algorithm to insert and retrieve the data into these tables was written. Table 3.1 below shows a snippet of this algorithm.

```
private static final String INSERT_QUERY="insert into,CustomerName,Address,City,AccountBalance)values
(2356,Teresa,Kisii,Nairobi,25000)";

private static final String INSERT_QUERY1="insert into Bankcustomers(CustomerID,CustomerName,Address,City

,AccountBalance)values (50, Susan,Nakuru,Nairobi,25000)";

private static final String SELECT_QUERY="select * from bankcustomers";

try {btm.begin();

for(int index = 1; index <= 5; index++) {

                                        pstmt.setInt(1,index);

                                        pstmt.setString(2, "Customers_" + index);

                                        pstmt.setString(3, "" + (4 + index));

                                        pstmt.setString(4, "Nairobi");

                                        pstmt.executeUpdate();      }

                                pstmt.close();

                                connection.close();

Connection connection =mySQLDS.getConnection(USER_NAME, PASSWORD);

PreparedStatement pstmt =connection.prepareStatement(INSERT_QUERY1);

for(int index = 1; index <= 5; index++) {

                                        pstmt.setInt(1,index);

                                        pstmt.setString(2, "Customers_" + index);

                                        pstmt.setString(3, "" + (4 + index));……

                                        pstmt.setString(4, "Nairobi");
```

Table 3.1: Snippet of the current Two Phase Commit Protocol

The above algorithm was compiles and run in *Jcreator IDE*.

As shown in the table, the algorithm consisted of three queries, two for inserting while the other one for retrieving records from the database. In this algorithm transaction partitioning is used. This consisted of the nested statements below;

try {…….}

catch {….}

Observation of this algorithm reveals that it consists of six of these nested statements, three for inserting and retrieving data and the three for error handling when errors are detected in the algorithm or the data resources. This is the root of the concurrency and blocking problems in two phase commit protocol. This is because these transactions are partitioned; hence each of them is transmitted to the data resources independent of each other. Hence if one of them fails to respond, the others are blocked, waiting for its recovery.

3.5 Simulation of a transaction Connectivity Based Clustering Algorithm

To address the concurrency and blocking problem in two phase commit protocol, the researcher designed and simulated transaction connectivity based clustering algorithm for optimizing distributed transactions.

Connectivity-based node clustering is an important network structure that can be employed in various ways to improve the quality of service of applications running on distributed system. A connectivity-based node clustering is the partitioning of network nodes into one or more groups based on their connectivity. In this setup, nodes were grouped into three sites: *NairobiBranch, HeadOffice* and *KisiiBranch.* However, the two branches (*NairobiBranch and KisiiBranch*) have connectivity to the *HeadOffice*. The communication among the partitioning was done by method calling.

3.5.1 Connectivity Based Transaction Clustering Algorithm Architecture

The algorithm consisted of the following components:

i) *Transaction manager*- the purpose of this component was to send and receive messages from the coordinator and participant. It also contains the recovery procedures to deal with transaction failures. *Bitronix Transaction Manager* was taken to be the transaction manager.

ii) *Coordinator*- its function is to monitor and execute atomic transactions. The Java main class, Coordinator was taken to be the coordinator, which was declared as follows:

```
public class Coordinator {
……….
}
```

iii) *Resource manager*- its function was to keep a record of stable committed transactions in storage. *MySQL* database was used for this perspective.

iv) *Resource Adapter*- The function of this component was to provide database connectivity. It was taken to be the *mySQL-connector-java-5.1.10-bin.jar* .

v) *Participants*-the function of these components was to take part in the voting process and take appropriate actions locally, which could be transaction commit or transaction abort. These were taken to be the three sub-transactions, two of which were to insert data into the database (*KisiiBranch* and *NairobiBranch*), while the remaining one was to update the *HeadOffice* database

vi) *Data managers*- the function of these components was to manage data transfer between its replica and other sites. These were taken to be the *Connection* constructs that provided the path to the databases. These were declared as follows:

```
Connection connection = DriverManager.getConnection(
        "jdbc:mySQL://localhost:3306/NairobiBranch", "root", "");
……….
.
Connection connection1 = DriverManager.getConnection(
        "jdbc:mySQL://localhost:3306/KisiiBranch", "root", "");
Connection connection2 = DriverManager.getConnection(
        "jdbc:mySQL://localhost:3306/HeadOffice", "root", "");
```

Figure 3.2 below shows the overall simulation architecture. Its shows the relationships among the above mentioned entities. As shown, the *bitronix transaction manager* directly communicates with the *coordinator*, which in turn communicates with *data managers*. The *data managers* communicate with *participants*. The *participants* communicate with the *resource managers* via the *resource adapter*.

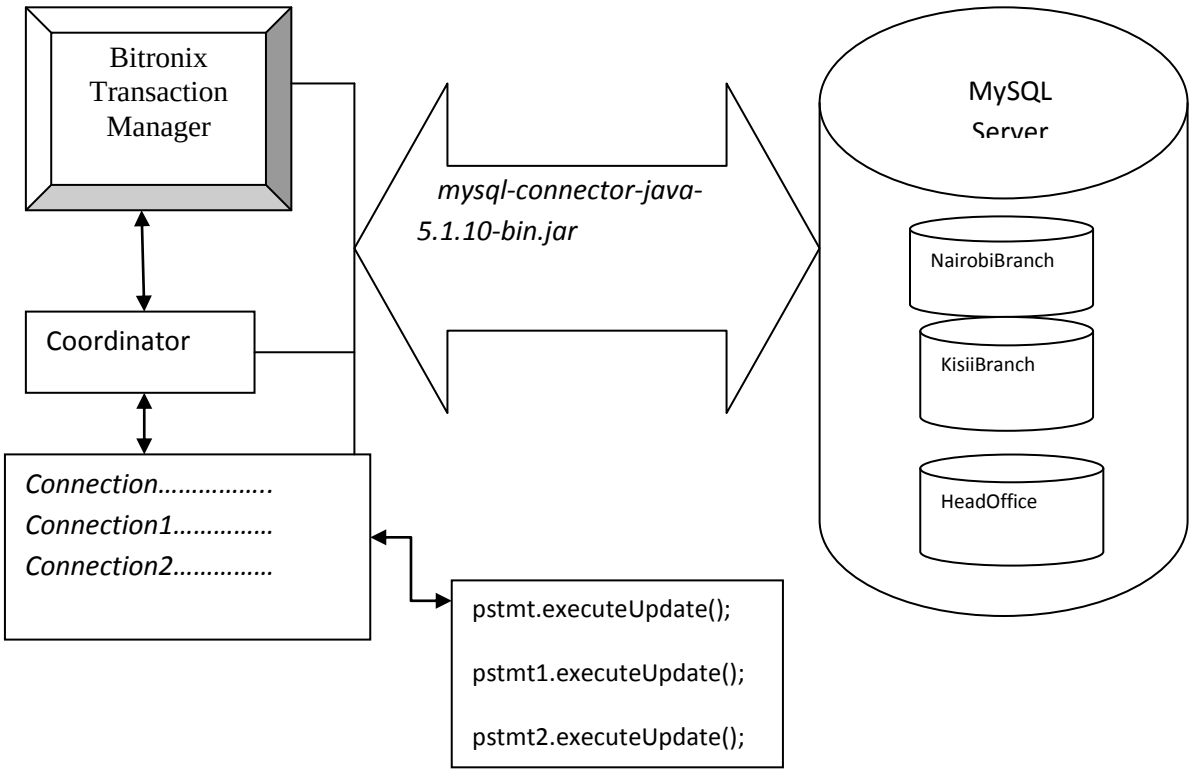

Fig 3.1 Overall simulation architecture

The diagram below shows the class diagram for the package *TwoPCCoordinator* package. *Coordinator* is the main class in the *TwoPCCoordinator* package. Its main function initiates *Coordinator* and *Participant* and to keep a list of them when a transaction occurs. The *Participant* thread will execute, redo or undo a sub-transaction. It carries out the proper proc
from the coordinator. It carries
recovery in case of failures.

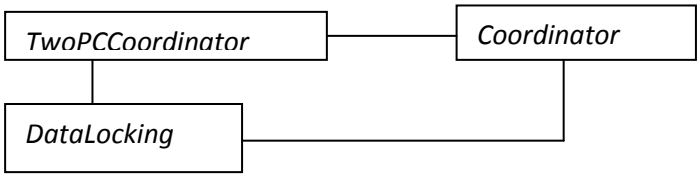

Figure 3.2 Transaction Clustering Algorithm Package Diagram

The *datalocking* component is responsible for locking data so that only one transaction have access to it at any particular moment. This is important if database inconsistency in distributed databases is to be avoided.

3.5.2 Procedure

1. Three databases were created in *MySQL* server. These were given names *KisiiBranch* and *NairobiBranch* and *HeadOffice*.
2. Three tables were created, one in each of these databases, with the name *bankcustomer.*
3. Table *bankcustomer* had five columns, namely *CustomerID, CustomerName ,Address, City and AccountBalance.*
4. The two phase commit protocol, with transaction clustering algorithm to insert the data into these tables was written. Table 3.2 below shows a snippet of this algorithm.

Table 3.2: Snippet of the Two-Phase Commit Protocol with Transaction Clustering

```
private static final String INSERT_QUERY="insert into
Bankcustomers(CustomerID,CustomerName,Address,City,AccountBalance)values (?,?,?,?,?)";

private static final String INSERT_QUERY1="insert into
bankcustomers(CustomerID,CustomerName,Address,City,AccountBalance)values (?,?,?,?,?)";

private static final String UPDATE_QUERY="Update bankcustomers SET AccountBalance='25000'";

BitronixTransactionManager btm =TransactionManagerServices.getTransactionManager();

    try {

          btm.begin();

Connection connection = DriverManager.getConnection(  "jdbc:mysql://localhost:3306/NairobiBranch", "root", "");

    PreparedStatement pstmt =connection.prepareStatement(INSERT_QUERY);

 Connection connection1 = DriverManager.getConnection(  "jdbc:mysql://localhost:3306/KisiiBranch", "root", "");

   PreparedStatement pstmt1 =connection1.prepareStatement(INSERT_QUERY1);

 Connection connection2 = DriverManager.getConnection(  "jdbc:mysql://localhost:3306/HeadOffice", "root", "");

 PreparedStatement pstmt2 =connection2.prepareStatement(UPDATE_QUERY);

  for(int index = 1; index <= 5; index++) {

System.out.println("-----------------------------------------------------------------------------------------------------");
System.out.println("NAIROBI_BRANCH_VOTE :TRANSACTION_COMMIT");

  //Sub-transaction-2: Inserting data into Table bankcustomers, residing in Database KisiiBranch

pstmt1.setInt(1,index);//Inserting data into first_column, CustomerID

pstmt1.setString(2, "Customer_" + index);//Inserting data into Second_column, CustomerName

 pstmt1.setString(3, "" + (4 + index));//Inserting data into Third_column,
Address……

……..

                                        pstmt1.setString(5, "25000");//Inserting data into
Fifth_column, AccountBalance

          pstmt1.executeUpdate();// Executing the INSERT_QUERY1

          pstmt1.close();//Terminating Sub-Transaction-2

          connection1.close();  //Terminating database connection for Sub-Transaction-2
```

The above algorithm was compiled and run in *Jcreator IDE*.

As shown in the table, the algorithm consisted of three queries, two for inserting while the other one for updating records from the database. It shows that transaction partitioning has not been used. This consisted of only one statement.

try {…….}

catch {….}

Observation of this algorithm reveals that it consists of only one branch of operation. Part one of this algorithm consist of sub-transactions that were meant to insert and update the databases, while the other part was for error handling. Hence there is a better concurrency control and sub-transactions do not block one another because all of them are executed at ago, simultaneously. Hence if they fail, they do so in a group and the transaction manager initiates a roll back as will be demonstrated in chapter four.

3.6 Data Collection and Analysis

In this research an experiment was designed to compare the performance of the transaction connectivity based clustering algorithm with the current Two-Phase Commit Protocol focusing on site and coordinator failure of distributed transactions. The data was collected using Bitronix transaction manager, which provided the required management feedback information on distributed transactions. It was then analyzed using the same software and the status of database manipulations.

CHAPTER FOUR

4.0 RESULTS DISCUSSION AND ANALYSIS

4.1 Introduction

This chapter gives an explanation for the results that were obtained from the experiment that was carried out. The probable explanations of the observed results and their analysis forms the basis of this chapter as discussed in the subsections below.

4.2 Analysis of distributed transaction failures in Two-Phase Commit(2PC) Protocol

The first objective of the researcher was to carry out an analysis of the two-phase commit protocol in order to identify its flaws. The researcher identified two major forms of 2PC failures, that is, site failure and coordinator failure.

The bitronix output below demonstrates a site failure. Table 4.0 below is a snippet of the bitronix transaction manager output that gives a clear indication of site failure.

…… Feb 5, 2015 6:27:39 AM bitronix.tm.BitronixTransactionManager logVersion……
at bitronix.tm.BitronixTransactionManager.<init>(BitronixTransactionManager.java:64)
at bitronix.tm.TransactionManagerServices.getTransactionManager(TransactionManagerServices.java:62)
at TwoPCProtocol.TwoPCCoordinatorFailure.main(TwoPCCoordinatorFailure.java:32)
Caused by: com.mysql.jdbc.exceptions.jdbc4.CommunicationsException: ***Communications link failure***
The last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets from the server.

Table 4.0: Demonstration of Two-Phase Commit Site Failure

The first line of this table gives an overview of the obtained results, that is, the information shown is the one contained in bitronix transaction log file. The second line is the initialization of the bitronix transaction manager, which has been given a unique ID of 64. The third line initializes the transaction manager services, and has been assigned a unique ID of 62. The '*TwoPCProtocol*', is the package name while '*TwoPCCoordinatorFailure*', is the name of the main class, which became our

coordinator. This was given a unique ID of 32. The next line gives information on the status of the communication link to the data resource. It is evident here that the data resource is down, as indicated by the statement, '*Caused by: com.mySQL.jdbc.exceptions.jdbc4.CommunicationsException: Communications link failure*'. The last line further explains that *Bitronix*, through the JDBC driver, which was '*mySQL-connector-java-5.1.10-bin.jar',* had sent a packet successfully. However, the data resource could not give any response. This was because the researcher intentionally put the *mySQL* server offline.

To demonstrate coordinator failure, the researcher put the data resource online and ran the source codes of coordinator failure involving distributed transactions directed towards a single data resource. Table 4.1 below gives a snippet of the source code for distributed transactions directed towards a single data resource.

```
……

public class TwoPCCoordinatorFailure {

private static final String DATABASE="TwoPCDistributedTransaction";

        private static final String USER_NAME="root";

        private static final String PASSWORD="";

        private static final String INSERT_QUERY="insert into

bankcustomers(CustomerID,CustomerName,Address,City)values (2356,Teresa,Kisii,Nairob, 25000)";

        private static final String SELECT_QUERY="select * from bankcustomers";

……..
```

Table 4.1: Coordinator Failure - Distributed Transactions and a Single Data Resource

As shown in the table, the first line is the main class that acts as a coordinator for the distributed transactions that are found within it. The next three lines initializes the

database, whose name was '*TwoPCDistributedTransaction*' , residing in *Wamp* Server, the user name, whose name was *'root'* , and the password of the user, which was set to be null. This was followed by the insert query, which if it committed, was to insert the values '*2356,Teresa,Kisii,Nairobi, 25000'*, into the table, whose name was *bankcustomers*, residing in database '*TwoPCDistributedTransaction*'. This was followed by another query to the same database, which if it could have committed, would have seen this transaction display all the records from the table named above.

The next table below gives a snippet of the output of the *Bitronix* transaction manager log file, where there is coordinator failure in distributed transactions and distributed data resource.

WARNING: error running recovery on resource 'TwoPCCoordinatoFailureClass1', ***resource marked as failed*** *(background recoverer will retry recovery)*
bitronix.tm.recovery.RecoveryException: cannot start recovery on a PoolingDataSource containing an XAPool of resource TwoPCCoordinatoFailureClass1 with 0 connection(s) (0 still available)
at bitronix.tm.resource.jdbc.PoolingDataSource.startRecovery(PoolingDataSource.java:227)
at bitronix.tm.recovery.Recoverer.recover(Recoverer.java:253)

Table 4.2: Coordinator Failure-Distributed Transactions and Distributed Data Resource

As shown in the table, the *Bitronix* transaction manager starts by obtaining the Java Virtual Machine unique ID, which is the address of the *localhost*, that is, 127.0.0.1. Line three of the snippet above clearly indicates that '*TwoPCCoordinatoFailureClass1*', which was our main class in the Java application and hence our coordinator, has failed. This is evident by *Bitronix* output statement, '***resource marked as failed*** *(background recoverer will retry recovery)*'.

4.2.1 Demonstration of Two-Phase Commit Coordinator and Site Failures

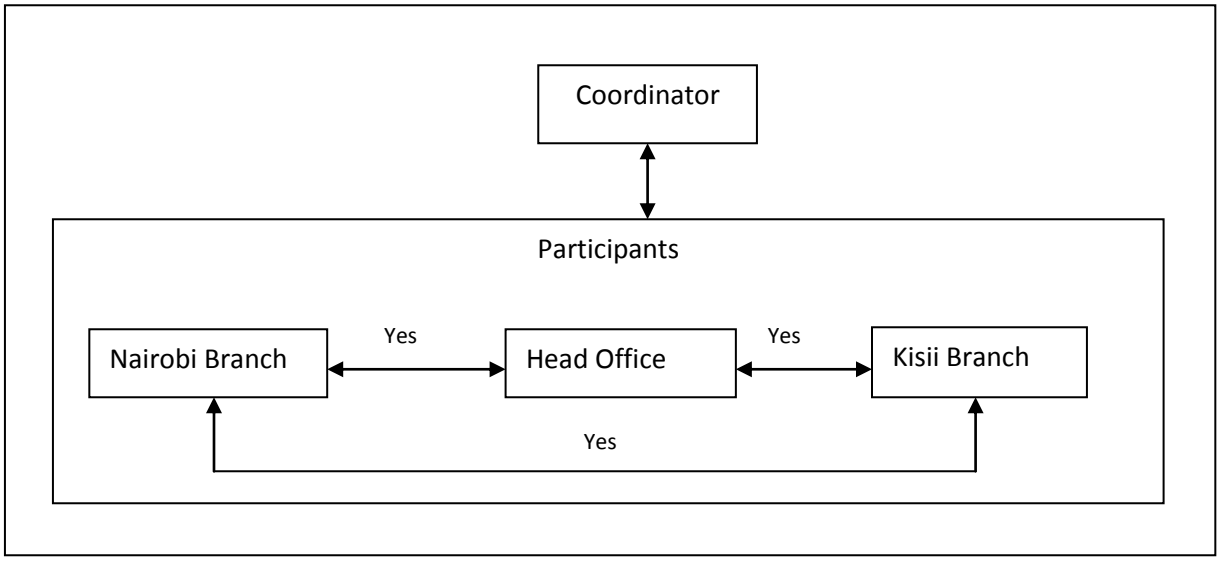

Fig 4.0: 2PC Coordinator & Site Failures

In Figure 4.0 above, suppose the coordinator sends COMMIT decision to Nairobi Branch, which upon receiving it, commits and unfortunately, goes down together with the coordinator. In such a scenario, even though Head office and Kisii Branch voted yes, they have to wait for either Nairobi branch or the coordinator to be up. They can't commit or abort, because they do not know the response of the Nairobi Branch (this decision was supposed to be communicated by the coordinator, which unfortunately, is also down). If that takes a long time (For example, a human must replace hardware), then availability suffers. This is why 2 phase commit is called a blocking protocol. 2PC suffers from allowing nodes to irreversibly commit an outcome before ensuring that the others know the outcome, too.

4.2.2 Demonstration of Three-Phase Commit

The 3PC turns 2PC into a non-blocking protocol – 3PC should never block on node failures as 2PC did. This protocol splits the commit/abort phase into two phases. It first communicates the outcome to every participant and then let them commit only after everyone knows the outcome.

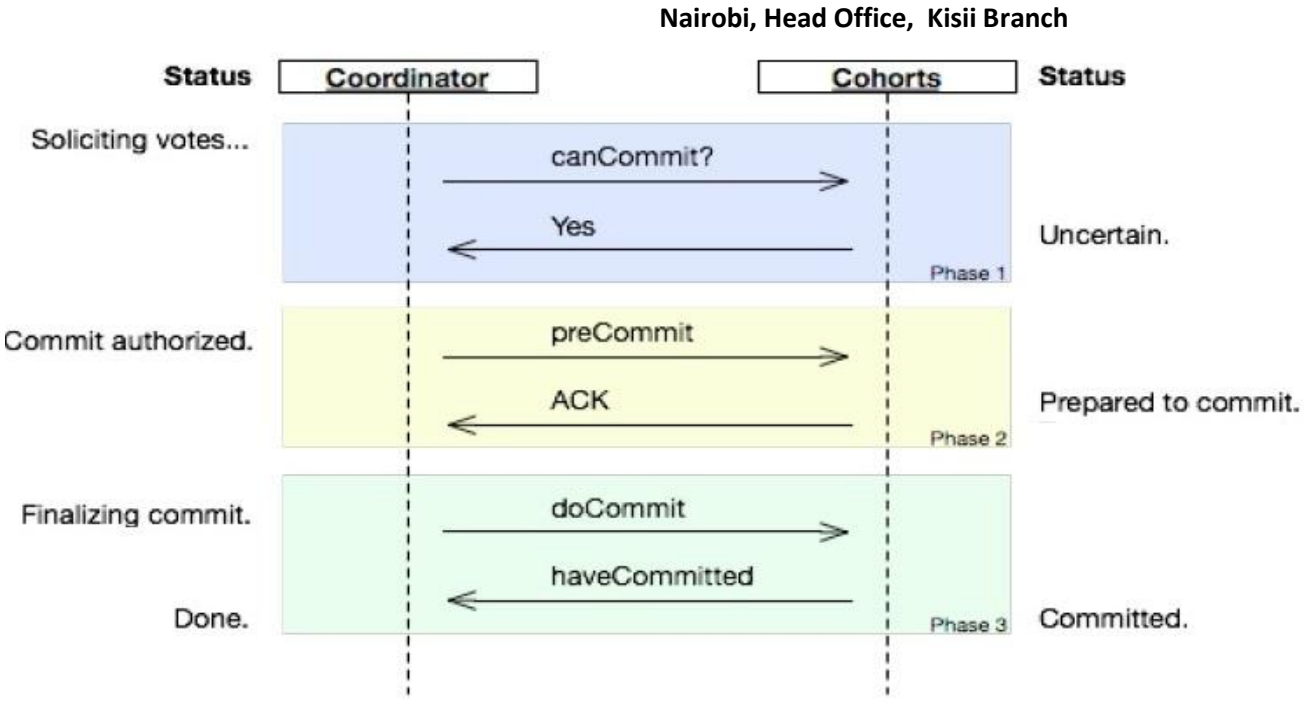

Fig.4.1 Three-Phase Commit overheads

Suppose the coordinator sends COMMIT decision to Nairobi Branch, which upon receiving it, commits and unfortunately, goes down together with the coordinator. In 3PC, if one of the three branches has received preCommit, they can all commit, meaning that Head office and Kisii Branch can go ahead and commit. If none of them has received preCommit, they can all abort. Therefore, 3PC doesn't block, it always makes progress by timing out. As evident above, there is so much overhead generated by 3PC compared to 2PC and this is the challenge for 3PC.

4.3 Simulation of a connectivity based transaction Clustering Algorithm

The second research objective was to simulate a transaction clustering algorithm for optimizing distributed transactions failure in two phase commit protocol. The results obtained indicate that site failure results were similar to the one obtained in Table 3.2.

The algorithm was modified such that all sites were TRUNCATED off their data except the *HeadOffice*, which remained with the data shown below and it was then run.

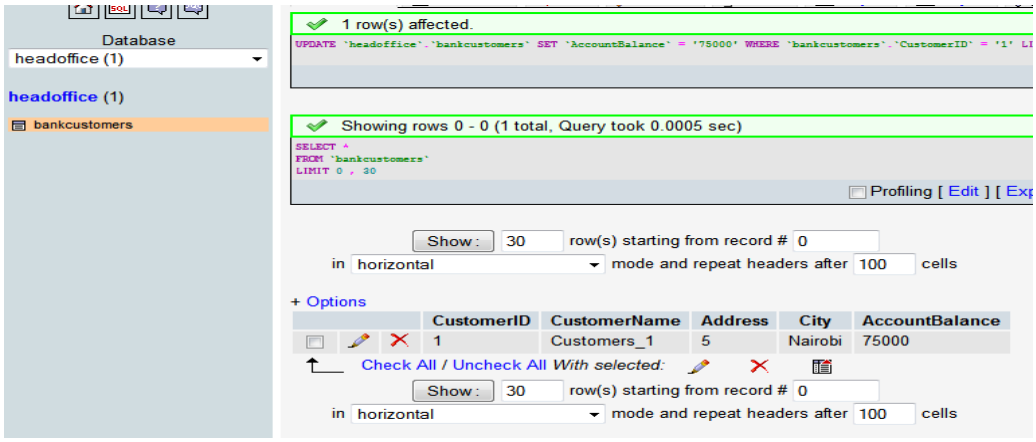

Table 4.3: HeadOffice Site Initial Status

This figure shows that *CustomerID* was 1, *CustomerName* was Customer_1, address was 5, City was Nairobi and *AccountBalance* was *75000*. The algorithm in Appendix V was meant to insert some data into the *KisiiBranch* and *NairobiNranch* databases, in their *bankcustomers* tables, while at the same time updating site *HeadOffice*, so that it could display *25000* as the AccountBalance.

This is evident from the snippet in Table 4.4 below.

```
private static final String INSERT_QUERY="insert into
Bankcustomers(CustomerID,CustomerName,Address,City,AccountBalance)values (?,?,?,?,?)";

private static final String INSERT_QUERY1="insert into
bankcustomers(CustomerID,CustomerName,Address,City,AccountBalance)values (?,?,?,?,?)";

private static final String UPDATE_QUERY="Update bankcustomers SET AccountBalance='25000'";

pstmt.setInt(1,index);//Inserting data into first_column, CustomerID

pstmt.setString(2, "Customer_" + index);//Inserting data into Second_column, CustomerName

pstmt.setString(3, "" + (4 + index));//Inserting data into Third_column, Address

pstmt.setString(4, "Nairobi");//Inserting data into Fourth_column, City

pstmt.setString(5, "25000");//Inserting data into Fifth_column, AccountBalance
```

Table 4.4: Snippet of the Insert and Update Transactions

Sure enough, when the algorithm was run, all these sites voted *TRANSACTION_COMMIT*, and the coordinator decision was, *GLOBAL_COMMIT*, as shown in Table 9 below. Note the lines:

NAIROBI_BRANCH_VOTE :TRANSACTION_COMMIT

KISII_BRANCH_VOTE :TRANSACTION_COMMIT

HEAD_PFFICE_VOTE :TRANSACTION_COMMIT

COORDINATOR_DECISION :GLOBAL_COMMIT

Table 4.5 below shows a Two-Phase Commit Protocol transaction clustering algorithm when all the three transactions have committed as a cluster.

```
----------Configuration: <Default>----------

Feb 15, 2015 8:39:42 PM bitronix.tm.BitronixTransactionManager logVersion

INFO: Bitronix Transaction Manager version 2.1.0

Feb 15, 2015 8:39:42 PM bitronix.tm.Configuration buildServerIdArray

WARNING: cannot get this JVM unique ID. Make sure it is configured and you only use ASCII characters. Will use IP
address instead (unsafe for production usage!).

Feb 15, 2015 8:39:42 PM bitronix.tm.Configuration buildServerIdArray

INFO: JVM unique ID: <127.0.0.1>

Feb 15, 2015 8:39:42 PM bitronix.tm.recovery.Recoverer run

INFO: recovery committed 0 dangling transaction(s) and rolled back 0 aborted transaction(s) on 0 resource(s) []
(restricted to serverId '127.0.0.1')

--------------------------------------------------------------------------------

NAIROBI_BRANCH_VOTE :TRANSACTION_COMMIT

--------------------------------------------------------------------------------

KISII_BRANCH_VOTE :TRANSACTION_COMMIT

--------------------------------------------------------------------------------

HEAD_PFFICE_VOTE :TRANSACTION_COMMIT

--------------------------------------------------------------------------------

COORDINATOR_DECISION :GLOBAL_COMMIT

--------------------------------------------------------------------------------

Feb 15, 2015 8:39:42 PM bitronix.tm.twopc.Preparer prepare

WARNING: executing transaction with 0 enlisted resource
```

Table 4.5 : Two Phase Commit Protocol Transaction Clustering Algorithm Output

A check on the three sites confined that these transactions had actually committed as shown in table 4.6, 4.7 and 4.8 below.

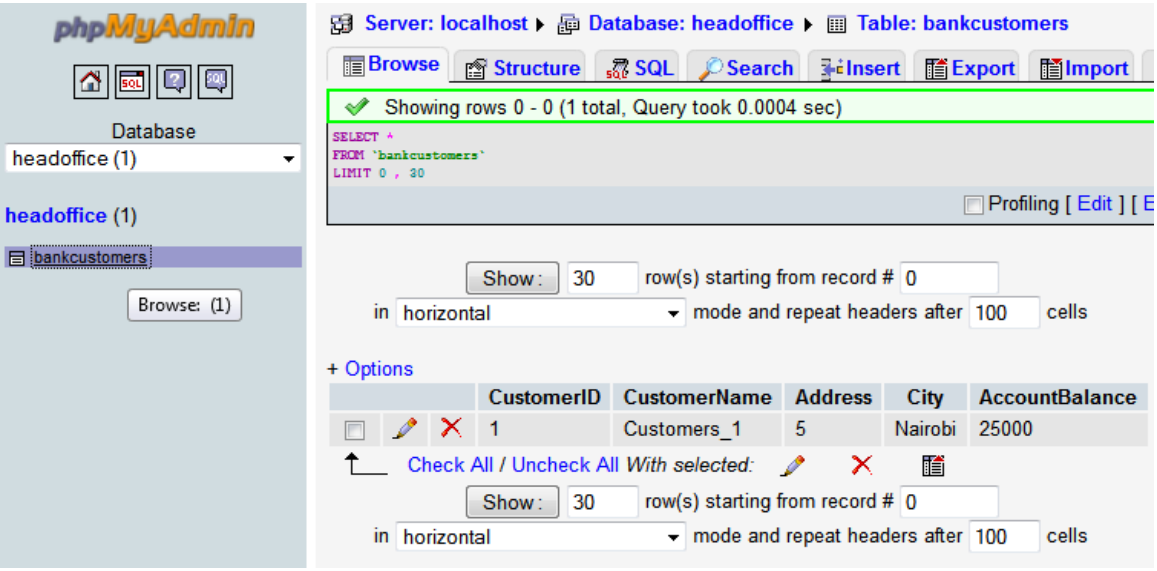

Table 4.6: HeadOffice Site Final Status After Commit

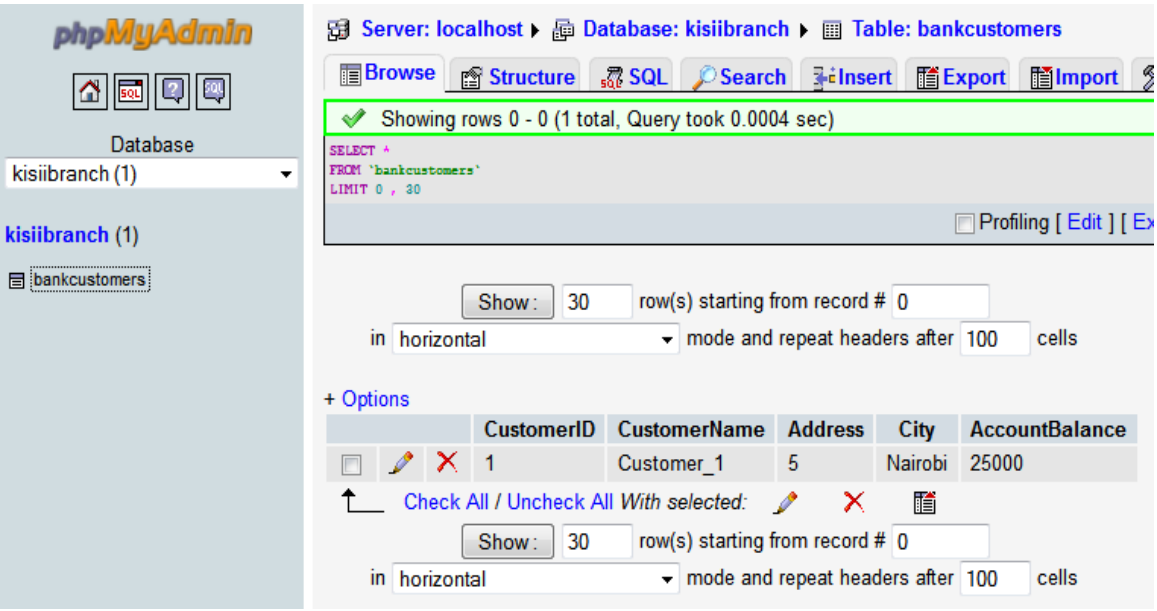

Table 4.7. KisiiBranch Site Final Status After Commit

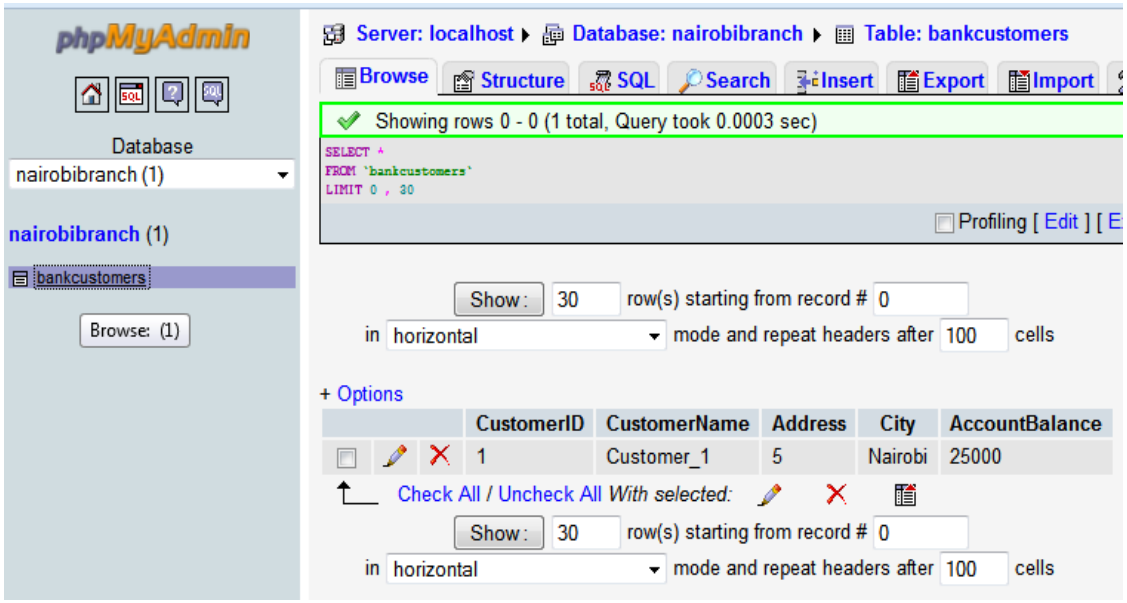

Table 4.8 . NairobiBranch Site Final Status After Commit

To investigate how this new algorithm handles site failures, one of the sites, the *NairobiBranch* was intentionally modified by setting the fifth column to be *AccountBalance1*, instead of *AccountBalance* as contained in the algorithm. Table 4.8.1 below shows how this modification was done.

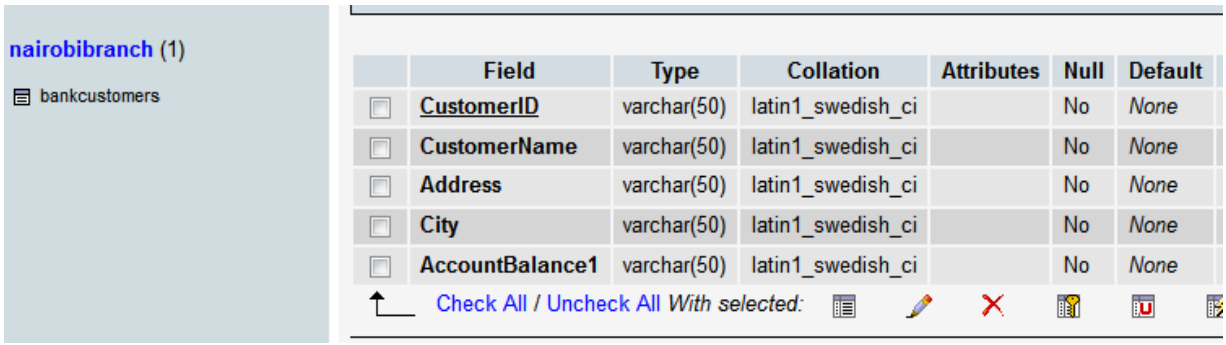

Table 4.8.1: NairobiBranch Modification Procedure

The algorithm was then re-run. Table 4.9 shows the *Bitronix* Transaction Manager output.

Table 4.9: Concurrency Control and Blocking in Transaction Clustering

WARNING: cannot get this JVM unique ID. Make sure it is configured and you only use ASCII characters. Will use IP address instead (unsafe for production usage!).

Feb 15, 2015 9:56:32 PM bitronix.tm.Configuration buildServerIdArray

INFO: JVM unique ID: <127.0.0.1>

Feb 15, 2015 9:56:32 PM bitronix.tm.recovery.Recoverer run

INFO: recovery committed 0 dangling transaction(s) and rolled back 0 aborted transaction(s) on 0 resource(s) [] (restricted to serverId '127.0.0.1')

Feb 15, 2015 9:56:32 PM bitronix.tm.BitronixTransactionManager shutdown

INFO: shutting down Bitronix Transaction Manager

Feb 15, 2015 9:57:32 PM bitronix.tm.BitronixTransaction timeout

*WARNING: transaction timed out: a Bitronix Transaction with GTRID [3132372E302E302E310000014B8E99657300000000], status=**MARKED_ROLLBACK,** 0 resource(s) enlisted (started Sun Feb 15 21:56:32 EAT 2015)*

The concurrency and blocking control in the new clustered algorithm is demonstrated in this table. If the coordinator poorly managed the transactions, the other two sites would have voted, *TRANSACTION_COMMIT*, since their sites were never affected by the changes that were carried out. However, none voted, and hence the developed algorithm can manage concurrency access to distributed databases. Moreover, since none of the sites voted *TRANSACTION_COMMIT*, they cannot claim to have been blocked by the failure of the site, *NairobiBranch.* Had they voted, *TRANSACTION_COMMIT*, they could have claimed to have been blocked by *NairobiBranch*, since they belonged to the same coordinator and therefore according to the two phase commit protocol requirement of atomicity, they were expected to commit together as a group.

4.3.1 Source code for Connectivity-Based Clustering algorithm

```
/*Author: Teresa Abuya, Msc (Computer Systems), Jkuat*/
        import java.sql.DriverManager;
        import java.sql.Connection;
        import java.sql.PreparedStatement;
        import java.sql.SQLException;
        import java.text.DateFormat;
        import java.text.SimpleDateFormat;
        import java.sql.Statement;
        import java.sql.Connection;
        import java.sql.PreparedStatement;
        import java.sql.ResultSet;
```

Java imports for database connectivity

```
            //Bitronix Transaction manager imports for database transactions monitoring and control
            import bitronix.tm.BitronixTransactionManager;
            import bitronix.tm.TransactionManagerServices;
            import bitronix.tm.resource.jdbc.PoolingDataSource;
//end of imports
             //Declaration of the class, "Coordinator"
                    public class Coordinator {
 //Constant constructs declaration and instantiation
                    private static final String DATABASE="KisiiBranch";
                    private static final String DB_DRIVER =
"com.mysql.jdbc.Driver";
                    private static final String DB_CONNECTION =
"jdbc:mysql://localhost:3306/customerdetails";
                    private static final String DB_USER = "root";
                    private static final String DB_PASSWORD = "";
                    private static final String USER_NAME="Teresa";
```

```
                    private static final String PASSWORD="Abuya";
//Database querries instantiation
private static final String INSERT_QUERY="insert into
Bankcustomers(CustomerID,CustomerName,Address,City,AccountBalance)values
(?,?,?,?,?)";
private static final String INSERT_QUERY1="insert into
bankcustomers(CustomerID,CustomerName,Address,City,AccountBalance)values
(?,?,?,?,?)";
private static final String UPDATE_QUERY="Update bankcustomers SET
AccountBalance='25000'";
//      Beginning of main class
        public static void main(String[] argv) {
        //Setting of the database parameters
                    PoolingDataSource mySQLDS=new PoolingDataSource();
        mySQLDS.setClassName("com.mysql.jdbc.jdbc2.optional.MysqlXADataSource";
                                    mySQLDS.setUniqueName("Coordinator");
                                    mySQLDS.setMaxPoolSize(3);
        mySQLDS.getDriverProperties().setProperty("databaseName", DATABASE);
        //Bitronix transaction manager initialization
                BitronixTransactionManager btm
=TransactionManagerServices.getTransactionManager();
 //Transaction Clustering for Coordinator and site failure circumvention
                try {
                        //Starting Bitronix transaction manager
            btm.begin();
  //Three Distributed databases are assumed, residing in Nairobi, Kisii and Head Office
 // Initializing JDBC Drivers and Database site NairobiBranch, with user as root
                Connection connection = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/NairobiBranch", "root", "");
```

```
        //Assigning INSERT_QUERRY to pstmt construct, an insert querry directed to
Nairobi site
            PreparedStatement pstmt =connection.prepareStatement(INSERT_QUERY);

  //Initializing JDBC Drivers and Database site KisiiBranch, with user as root
            Connection connection1 = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/KisiiBranch", "root", "");

       // Assigning INSERT_QUERRY1 to pstmt construct, an insert querry directed to
Kisii site
            PreparedStatement pstmt1
=connection1.prepareStatement(INSERT_QUERY1);

          // Initializing JDBC Drivers and Database site KisiiBranch, with user as root
            Connection connection2 = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/HeadOffice", "root", "");
                    //Assigning UPDATE_QUERY to pstmt construct, an update querry
directed to Head office site
            PreparedStatement pstmt2
=connection2.prepareStatement(UPDATE_QUERY);

        //The for--loop, acting as a Data Manager
      for(int index = 1; index <= 5; index++) {

                    //Sub-transaction-1: Inserting data into Table bankcustomers, residing in
Database NairobiBranch

pstmt.setInt(1,index);//Inserting data into first_column, CustomerID
```

```
pstmt.setString(2, "Customer_" + index);//Inserting data into Second_column,
CustomerName
pstmt.setString(3, "" + (4 + index));//Inserting data into Third_column, Address
pstmt.setString(4, "Nairobi");//Inserting data into Fourth_column, City
pstmt.setString(5, "25000");//Inserting data into Fifth_column, AccountBalance
pstmt.executeUpdate();// Executing the INSERT_QUERY
 pstmt.close();//Terminating Sub-Transaction-1
 connection.close();//Terminating database connection for Sub-Transaction-
     //Status of the voting in Database located at site, NairobiBranch
          System.out.println("-------------------------------------------------------------------
----------------------------");
                    System.out.println("NAIROBI_BRANCH_VOTE
:TRANSACTION_COMMIT")
    //Sub-transaction-2: Inserting data into Table bankcustomers, residing in Database
KisiiBranch
pstmt1.setInt(1,index);//Inserting data into first_column, CustomerID
pstmt1.setString(2, "Customer_" + index);//Inserting data into Second_column,
CustomerName
 pstmt1.setString(3, "" + (4 + index));//Inserting data into Third_column, Address
pstmt1.setString(4, "Nairobi");//Inserting data into Fourth_column, City
pstmt1.setString(5, "25000");//Inserting data into Fifth_column, AccountBalance
pstmt1.executeUpdate();// Executing the INSERT_QUERY1
 pstmt1.close();//Terminating Sub-Transaction-2
connection1.close();  //Terminating database connection for Sub-Transaction-2
//Status of the voting in Database located at site, KisiiBranch
                          System.out.println("-------------------------------------------------
-------------------------------------------------");
 System.out.println("KISII_BRANCH_VOTE :TRANSACTION_COMMIT");
```

```
//Sub-transaction-3: Update data in Table bankcustomers, residing in Database HeadOffice
                pstmt2.executeUpdate();// Executing the UPDATE_QUERY
            pstmt2.close();//Terminating Sub-Transaction-3
            connection2.close(); //Terminating database connection for Sub-Transaction-3
//Status of the voting in Database located at site, HeadOffice
                        System.out.println("-----------------------------------------------------------
----------------------------------------");
            //
System.out.println("HEAD_PFFICE_VOTE :TRANSACTION_COMMIT");
                        System.out.println("-----------------------------------------------------------
---------------------------------------");
             //Status of the Coordinator Decision
System.out.println("COORDINATOR_DECISION :GLOBAL_COMMIT");
                        System.out.println("-----------------------------------------------------------
---------------------------------------");
btm.commit();//Transaction Manager Commits the Group of 3-Sub-Transactions
                            System.out.println("-----------------------------------------------------
-------------------------------------------");
 btm.rollback();//Transaction Manager rollsback  the Group of 3-Sub-Transactions
          }
            //Error_Handling if commit of the 3-Sub-Transactions is not possible
                                }catch (Exception ex)
                            {
                            //ex.printStackTrace();
                            btm.shutdown();//Transaction Manager is ShutDown
System.out.println("-----------------------------------------------------------------------------
----------------");        }}}
```

The above algorithm explains how a clustering algorithm is used to improve the performance of 2PC.

To demonstrate the working of a clustering algorithm in Two-Phase Commit Protocol, the above code was inserted with relevant fields like updating the three databases with the same name such that they read the same information.

To test their working all the five fields namely CustomerID, Costomer Name,address, City and account balance was fed with information ,but initially they were inconsistent with Nairobibranch and KisiiBranch having no account balance while headquarters had accountbalance of Kshs. 7500.

This was achieved by creating an algorithm that would INSERT_QUERY in the two branches.Thus Nairobibranch and Kisii branches used INSERT-QUERY AND INSERT-QUERY1 to insert data to the fields and accountbalance to read Kshs.25000. UPDATE_QUERY was used for headoffice to change the current balance of Kshs.75000 to Kshs.25000.

For all the databases to read the same data information…the algorithm was run and all the databases had the same account balance of Kshs.25000.

After running the file at bironix output indeed showed that transactions were COMMTTED and all of them voted and it resulted to coordinator decision making a GLOBAL COMMIT message as shown in the snippet below:-

```
INFO: recovery committed 0 dangling transaction(s) and rolled back 0 aborted transaction(s) on 0 resource(s) [] (restricted to serverId '127.0.0.1')

--------------------------------------------------------------------------------

NAIROBI_BRANCH_VOTE :TRANSACTION_COMMIT

--------------------------------------------------------------------------------

KISII_BRANCH_VOTE :TRANSACTION_COMMIT

--------------------------------------------------------------------------------

HEAD_PFFICE_VOTE :TRANSACTION_COMMIT

--------------------------------------------------------------------------------

COORDINATOR_DECISION :GLOBAL_COMMIT

--------------------------------------------------------------------------------
```

From the above snippet it shows that all transactions agrred to vote as a group and voted COMMIT therefore resulting to a GLOBAL-COMMIT.The above output was used to demonstrate atomicity whereby only one table was used for all the three sites for demonstration of what happens when a transaction is carried out on one person's account. The Bankcustomer table had five fields namely CustomerID, Costomer Name,address, City and Accountbalance with the same information across all sites,hence the GLOBAL –COMMIT decision.

Had any of the sites or all of the sites failed to COMMIT the decision from the coordinator would have been a GLOBAL-ABORT and the transactions would have been "marked-rollback" by the birtonix transaction manager as shown in the snippet below.

Feb 15, 2015 9:56:32 PM bitronix.tm.BitronixTransactionManager shutdown

INFO: shutting down Bitronix Transaction Manager

Feb 15, 2015 9:57:32 PM bitronix.tm.BitronixTransaction timeout

*WARNING: transaction timed out: a Bitronix Transaction with GTRID [3132372E302E302E310000014B8E99657300000000], status=**MARKED_ROLLBACK,** 0 resource(s) enlisted (started Sun Feb 15 21:56:32 EAT 2015)*

This explains the clustering algorithm such that all sites agree to commit then they commit as a group and if any of the sites has inconsistent data then all sites will agree to rollback.This improves the efficiency of the current 2PC.

It is important to note that in a clustering algorithm all the transactions are grouped together in one single coordinator and the coordinator decides whether to commit or abort a transaction depending on the decision at participant's site. The transactions commit or abort as a group.

The benefit of this is that it minimizes blocking of transactions that is the case in 2PC whereby they keep waiting for coordinator to give direction on the next course of action. If the coordinator is down participants will keep waiting until its recovery. This is

inefficient management of transactions. This has been overcome by making a common decision as a group either to commit or abort. This does not leave participants in uncertain state as demonstrated in fig 4.5.However in the current two-phase it is not clear how two-phase commit is done as bitronix transaction manager only reports a shutdown and no further information is provided as shown in appendix 1.

4.3.2 Connectivity based clustering algorithm in Two-Phase Commit Protocol

The approach adopted was the clustering of the nodes based on their connectivity. This was done to address the extra overheads generated in 3PC while at the same time addressing the blocking problem in 2PC caused by either coordinator or site failure, which occasion network partitioning. The participants were designed as sub-classes and the coordinator as the main Java class. The transaction manager was the Bitronix transaction manager (BTM).

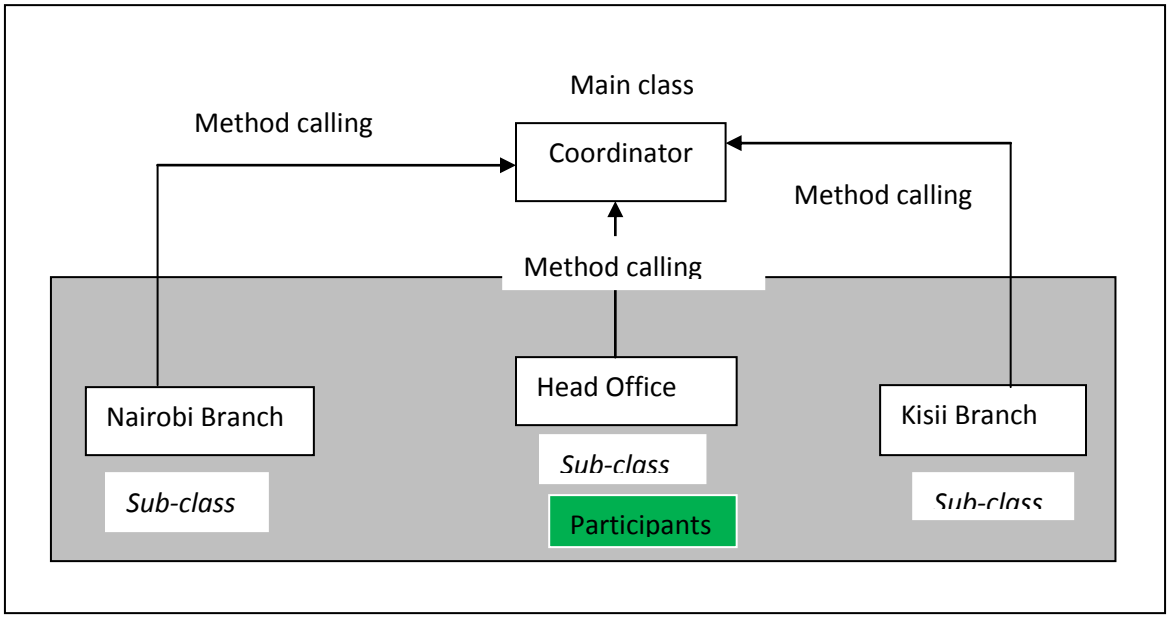

Fig.4.2: Connectivity based clustering algorithm

In this approach, Java was used as the programming language because of its support for inheritance and polymorphism, a key concept that was needed in distributed transactions

handling. The sub-classes were designed for the three branches with the main class being the coordinator. The concept of method calling was used by the main class to execute the relevant participant transaction. Inheritance and polymorphism ensured that the coordinator decisions were implemented differently by the participants (for example the updating of different customer tables in distributed databases-Kisii, Nairobi and Head office). In this arrangement, the participants need not wait for the other site(s) or coordinator that is down. Instead, they either perform GLOBAL_COMMIT or GOBAL_ABORT depending on the participant votes. This reduces communication overheads due to the utilization of method calling and polymorphism which reduces replication of the implementation process. In so doing, this approach brings in the 3PC concept of non-blocking as a result of site or coordinator failure while at the same time addressing the challenges of communication overheads.

The sub-section that follow give illustrations of how the model components represented above were employed to achieve connectivity-based clustering algorithm two phase commit protocol for optimizing distributed transactions failure.

4.3.3 Setup

Three laptop computers and an Ethernet switch were used to simulate the developed connectivity-based clustering algorithm two phase commit protocol for optimizing distributed transactions failure. The connections were done as demonstrated in Figure 4.3 below.

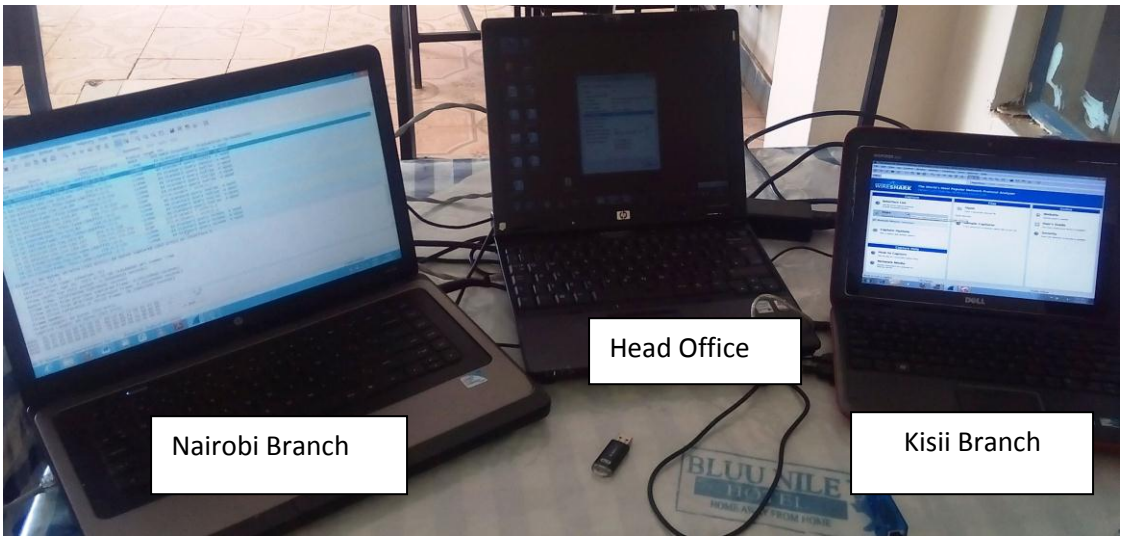

Figure 4.3: Practical Implementation

As this figure above shows, one machine was designated as the Head office , another one was designated as the Nairobi branch while the remaining one was designated as the Kisii branch. The transaction manager, coordinator, resource adapter and resource manager were all running in the head office, while data managers were executing in each of the three branches. The procedures below were then used to achieve the efficient distributed transactions handling.

4.4 Comparison of the transaction clustering algorithm with the current 2PC

To carry out a performance comparison between the current two phase commit protocol and the developed transaction clustering algorithm, six parameters were used. These included concurrency, blocking, coordinator failure, site failure, inconsistency and efficiency.

Concurrency control was achieved in the new algorithm because it was demonstrated by the results in Table 4.5 and Table 4.9 above. In table 4.5, all the sites voted *TRANSACTION_COMMIT*, while in Table 4.9, none of the sites voted *TRANSACTION_COMMIT*, implicitly meaning that they voted TRANSACTION_ABORT. In fact, the line,

WARNING: transaction timed out: a Bitronix Transaction with GTRID [3132372E302E302E310000014B8E99657300000000], status=MARKED_ROLLBACK, 0 resource(s) enlisted (started Sun Feb 15 21:56:32 EAT 2015)

Stresses this point, that is, the Bitronix Transaction Manager initiated a rollback. With the current two phase commit protocol, it is not clear whether the sites voted *TRANSACTION_COMMIT* or otherwise. The Bitronix Manager only indicates a shutdown and no further information is given concerning the fate of these transactions. One has therefore to manually check the databases to establish what transpired. This observation can be made in the snippet taken below:

at bitronix.tm.resource.common.XAPool.getConnectionHandle(XAPool.java:123)

at bitronix.tm.resource.jdbc.PoolingDataSource.startRecovery(PoolingDataSource.java:223)

... 6 more

Feb 15, 2015 8:01:14 PM bitronix.tm.BitronixTransactionManager shutdown

INFO: shutting down Bitronix Transaction Manager

Process completed.

This is a poor management of concurrent processes. There would have been a communication whether it was a rollback, a commit or an abort.

Blocking was avoided in the new algorithm by having transactions clustered together instead of partitioning them by use of the nested statement as evidenced in above algorithms.

Try {……

...} catch {……

...}

When nested statements are used, there is a probability of one or more of the sites committing or aborting independently. That is, the nested statement may turn true and proceed to commit. Hence if rollback is not initiated properly, the other sites will be blocked by this inconsistent state of this site, in their quest to obey the two phase commit protocol requirement of atomicity. As it has already been stated, in the new algorithm, all the transactions are clustered together, so that they either commit or abort as a single entity, in line with the 2PC requirement of atomicity.

Moreover, efficiency is improved with the new transaction clustering approach. By having sites taking a common decision via communication from the coordinator, there is no need to wait for sites that are down or network partitioned. This was demonstrated by the uniform TRANSACTION_COMMIT decisions and the implicit TRANSACTION_ABORT. However, in the current 2PC that was demonstrated, it was not clear how the voting happened; Bitronix transaction manager only reported a shutdown and no further information is provided.

This could cause inefficiency as users need to struggle trying to figure out what might have happened as even Bitronix never reported a rollback. Databases have to be checked to see if commits or rollbacks happened. This is a typical inefficiency.

CHAPTER FIVE

5.0 SUMMARY CONCLUSION AND RECOMMENDATIONS

5.1 Introduction

This chapter provides the summary, conclusions and recommendations emanating from the research study findings. The possible future research areas in this field are also given, which are believed, if properly implemented will seal the research gaps that were identified in this research study.

5.2 Summary

The research on the simulation of a two phase commit protocol clustering algorithm for optimizing distributed transaction failure was a success. The research objectives were: To analyze the distributed transaction failures in two-phase commit protocol; to simulate a transaction clustering algorithm for optimizing distributed transactions failure in two phase commit protocol; and to compare the performance of the transaction clustering algorithm with the current two phase commit protocol. All these objectives were effectively achieved. To start with, distributed transaction failures in two-phase commit protocol were demonstrated using Bitronix transaction manager. It was observed that both site failure and coordinator failures are common in two-phase commit protocol. Site failure occurred when a site was intentionally brought down so that it does not provide in the voting process.

On the other hand, coordinator failure occurred when the coordinator poorly managed the various transactions that it was meant to monitor to ensure their successful commit or rollback. Secondly, a transaction clustering algorithm for overcoming distributed transactions failure in two phase commit protocol was simulated using *Jcreator IDE* and Bitronix transaction manager. It was shown to successfully solve the problem with the current two-phase commit protocol, in that the sites committed and aborted as a group, which is in line with the objectives of the two-phase commit protocol. It was shown that when one site is down, the rest of the sites implicitly vote quit, hence no database modification happens. However, when all sites are up, each one of them voted and the three transactions committed successfully, modifying the database contents. With the

clustering algorithm, the coordinator was efficient in its transactions, in that all the transactions either committed or aborted as a group. The last objective was to compare the performance of the transaction clustering algorithm with the current two phase commit protocol. It was shown that with the clustering algorithm, all the transactions are grouped under a single coordinator. With this mechanism, all the transactions either commit or rollback as a group. However with the current two phase commit protocol, the transactions are partitioned in that in some situations, the various transactions may commit as others abort, hence bringing on the problem of inconsistence.

The experimental research design, which was adopted for this study, proved successful. It involved the practical design of the transaction clustering algorithm, using *Jcreator IDE*. This algorithm was simulated, with *mySQL* acting as the backend data manager. The data was Bitronix transaction manager, which provided the required management feedback information on distributed transactions. It was then analyzed using the same software and the status of database manipulations. The results obtained indicated that by using the proposed algorithm, the site and coordinator transactions failures associated with the current Two-Phase Commit can be sreduced. This was achieved by eliminating transaction partitioning, which is an inherent feature of the current two phase commit protocol. In a partitioned environment, blocking caused by the failure of the coordinator when participants are in uncertain state is a common problem.

To counter this, the researcher clustered all the sub-transactions in a single sub-class and used the principle of inheritance to obtain variables and methods from the main super-class, which was the coordinator, by message passing and message calling. The transaction manager was then employed to coordinate the execution activities of the coordinator. Transaction commit or transaction roll back was then reported by the transaction manager. By suing this approach, all the transactions in the sub-class either commit in their entirety or fail in their entirety, which is in line with the principles of a two phase commit protocol.

5.3 Conclusions

This research focused on site and coordination failures as common drawbacks in the two-phase commit protocol. The developed algorithm had four distinct steps. The first step was for the Coordinator to send a *VOTE_REQUEST* message to all participants, in this case, all the data resources that were to be manipulated by the coordinator transactions. When a participant received a *VOTE_REQUEST* message, it returned either a *VOTE_COMMIT* message to the coordinator telling the coordinator that is prepared to commit or a *VOTE_ABORT* message. Each participanst that voted for a commit waited for the final reaction by the coordinator. If a participant received a *GLOBAL_COMMIT* message, it locally committed the transaction else it aborted the local transaction. The commit process was characterized by database update while the abort process left the database unaltered, in accordance with the atomicity principle. All these were accomplished in *Jcreator IDE* and *mySQL* database. The research objectives were achieved as already stated above. The new algorithm had an improved performance in as far as site failure and coordinator failures were concerned. It was shown that all transactions committed or aborted and the databases were left in a consistency state after a commit, or in unchanged state in case of a transaction abort process.

5.4 Recommendations and future works

Coordinator and site failure are common problems in the current two phase commit protocol. As such, several efforts have been made to overcome them. The design of the three phase commit protocol was meant to overcome these challenges by introducing an extra phase, called the pre-commit phase. However, this approach is complicated to implement, has more communication overheads and maintaining inconsistency towards network partitioning is a serious problem. The developed algorithm has been shown to solve the coordinator failure and site failures without introducing extra overheads, which is a serious problem in three phase commit protocol. Moreover, the algorithm has been shown to be ideal in maintaining consistency of the database and chances of partitioning are rare because all transactions are clustered and hence either commit or rollback as a

group. Therefore the researcher recommends it adoption in distributed transaction handling. The possible improvement areas include the design of this algorithm so that the explicit *TRANSACTION_COMMIT* voting can be part of the responses received from the participants. Moreover, there is need to implement this algorithm in other backbends, such as *SQL* and oracle servers.

REFERENCES

Al-Houmaily, J. & Yousef , P. K.(2004). Chrysanthis:1-2 PC: *The one-two phase atomic commit protocol*.

Amir, Y., Coan, B.A., Kirsch, J.& Lane, J.(2010). *Byzantine Replication under Attack*, Proc. IEEE Int'l Conf. *Dependable Systems and Networks*, pp. 105-144.

Andrew, S. & Tanenbaum, M. (2006). Distributed Systems: Principles and Paradigms,2nd ed. Maarten Van Steen.

Byun, T. & Moon, S.(2012). Non-blocking two-phase commit protocol to avoid unnecessary transaction abort for distributed systems. Cheongryang 130-012 Seoul South Korea. *Journal of Systems Architecture,* 43(5), 245-254.

Cowling, J., Myers, D., Liskov, B., Rodrigues, R. & Shri, L.(2010). HQ Replication: *A Hybrid Quorum Protocol for Byzantine Fault Tolerance*, Proc. Seventh Symp. *Operating Systems and Implementation*, pp. 177-190.

Dollimore, J. G. Coulouris,, Kindberg T. (2007)"*Distributed systems – Concepts and Design",* fourth edition.

Fahd,S. Tarek, H. & Al-Otaibi, (2011). Dynamic Load-Balancing Based on a *Coordinator and Backup Automatic Election in Distributed Systems*", International Journal of Computing and Information Sciences 9(1).

Gray, J. and Reuter, M.(2001)Minimizing the Number of Messages in Commit Protocols. Worhahop on Fundamental 1.v dues in Distributed Computing, Pala Mesa. 90-92.

Goebel, V. (2011) *Distributed Database Systems*, Department of Informatics, University of Oslo.

Groote, F., & Willemse, T. A. C., (2005). *Parameterized Boolean equation systems,Theor.Comput. Sci.,* 343(3), 332–369

Groote, F., Mathijssen, A. ,Weerdenburg, M. & Usenko, S.(2008) From μCRL to mCRL2: *motivation and outline Electr. Notes Theor.Comput. Sci.*, 162, 191–196.

Jamuna, P., Sathiyadevi, S. & Tamilarasi, S. (2012). Backup Two Phase Commit Protocol(B2PC)renders trustworthy coordination problem over distributed transactions, *International Journal for Advanced Research in Computer Science and Software Engineering,* 2(9).

Kitsuregawa, M. & Reddy, K. (2006) Reducing the blocking in two-phase commit with backup site. *Information Processing Letters,* 86(1), 39-47.

Kotla, R., Alvisi, L., Dahlin,M., Clement, A. & Wong, E. (2010). Zyzzyva: *Speculative Byzantine Fault Tolerance,* Proc. 21st ACM Symp. Operating Systems Principles,pp. 45-58,Oct.2010.

Kumar, A, Y. & Ajay A. (2011). *A Distributed Architecture for Transactions Synchronization in Distributed Database Systems*, *International Journal on Computer Science and Engineering* 2(6).

Lomet, D. and Salzberg, B.(2003) Transaction-tirnc Databases. In Temporal Databases: Theory, UC- sign, and Implementation.89-99.

Manikandan, V., Ravichandran, R., Suresh, R., & Sagayaraj, F.(2012). *An efficient Non-blocking Two Phase Commit Protocol for Distributed Transactions* 2(3), 778-791.

MengQingyuan, W. & Haiyang, X. (2008). A New Model for Maintaining Distributed Data Consistence, *International Conference on Computer Science and Software Engineering, IEEE* 6(14).

Parul, Y., Singh,P., Amal S. & Sanchit L.(2011). An Extended Three Phase Commit Protocol for concurrency control in Distributed Systems. *International Journal for computer applications,* 21(10) .

Peng-Yeng Yin, P., Shiuh-Sheng, Y., Pei-Pei, W. & Yi-Te, W. (2006). A hybrid particle swarm optimization algorithm for optimal task assignment in distributed systems. *Computer Standards & Interfaces,* 28(4), 441-450.

Samaras, G., Britton, K., Citron, A., and Mo- han, C.(2003) Two-Phase Commit Optimirations and Tradeoffs in the Commercial Environment. Proc. Data Engineering Conference, Vienna,67-90.

Schapiro, R. & Milistein, R. (2012). *Failure recovery in a distributed database system,*" in Proc. 1978COMPCONConf.,23(6)213-310

Syam Menon, S.(2005). Allocating fragments in distributed databases. IEEE *Transactions on Parallel and Distributed Systems*16, 577-585.

Taibi, T., Abdelouahab, A., Wei Jiann, L., Yeong, F.& Ting Ng, C. (2006). Design and Implementation of a Two-Phase Commit Protocol Simulator, *The international Arab Journal of Information Technology,* 3(1).

Tanenbaum A, Van Steen M. (2007). Distributed systems – Principles and Paradigms, 2nd ed.Armsterdam, The Netherlands: John Wiley and sons.

Taranum, F. Tabassum,K. & Damodaram A (2011). *A Simulation of Performance of Commit Protocols in Distributed Environment.* in PDCTA, CCIS 203, pp. 665–681, Springer.

APPENDICES

Appendix I:Current 2PC code to insert and retrieve data from the database

```
package TwoPCCoordinator;

 /*
 Author: Teresa Abuya, Msc (Computer Systems), Jkuat
 */
//Java imports for database connectivity
        import java.sql.DriverManager;
        import java.sql.Connection;
        import java.sql.PreparedStatement;
        import java.sql.SQLException;
        import java.text.DateFormat;
        import java.text.SimpleDateFormat;
        import java.sql.Statement;
        import java.sql.Connection;
        import java.sql.PreparedStatement;
        import java.sql.ResultSet;
//Bitronix Transaction manager imports for database transactions monitoring and control
                import bitronix.tm.BitronixTransactionManager;
                import bitronix.tm.TransactionManagerServices;
                import bitronix.tm.resource.jdbc.PoolingDataSource;
//end of imports
                 //Declaration of the class, "Coordinator"
        public class Coordinator {
 //Constant constructs declaration and instantiation
```

```
                    private static final String DATABASE="KisiiBranch";

                    private static final String DB_DRIVER = "com.mysql.jdbc.Driver";

                    private static final String DB_CONNECTION =
"jdbc:mysql://localhost:3306/customerdetails";

                    private static final String DB_USER = "root";

                    private static final String DB_PASSWORD = "";

                    private static final String USER_NAME="Teresa";

                    private static final String PASSWORD="Abuya";

        //Database querries instantiation

                        private static final String INSERT_QUERY="insert into
Bankcustomers(CustomerID,CustomerName,Address,City,AccountBalance)values (?,?,?,?,?)";

                        private static final String INSERT_QUERY1="insert into
bankcustomers(CustomerID,CustomerName,Address,City,AccountBalance)values (?,?,?,?,?)";

                        private static final String UPDATE_QUERY="Update
bankcustomers SET AccountBalance='25000'";


//      Beginning of main class


        public static void main(String[] argv) {

            //Setting of the database parameters

                PoolingDataSource mySQLDS=new PoolingDataSource();

mySQLDS.setClassName("com.mysql.jdbc.jdbc2.optional.MysqlXADataSource");

mySQLDS.setUniqueName("Coordinator");

mySQLDS.setMaxPoolSize(3);
        mySQLDS.getDriverProperties().setProperty("databaseName", DATABASE);

        //Bitronix transaction manager initialization
```

```
                BitronixTransactionManager btm
=TransactionManagerServices.getTransactionManager();
  //Transaction Clustering for Coordinator and site failure circumvention
                try {
           //Starting Bitronix transaction manager
            btm.begin();
  //Three Distributed databases are assumed, residing in Nairobi, Kisii and Head Office
  // Initializing JDBC Drivers and Database site NairobiBranch, with user as root
                Connection connection = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/NairobiBranch", "root", "
          //Assigning INSERT_QUERRY to pstmt construct, an insert querry directed to Nairobi
site
            PreparedStatement pstmt =connection.prepareStatement(INSERT_QUERY);
              //Initializing JDBC Drivers and Database site KisiiBranch, with user as root
            Connection connection1 = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/KisiiBranch", "root"
        // Assigning INSERT_QUERRY1 to pstmt construct, an insert querry directed to Kisii site
            PreparedStatement pstmt1 =connection1.prepareStatement(INSERT_QUERY1);
                              //


           // Initializing JDBC Drivers and Database site KisiiBranch, with user as root
            Connection connection2 = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/HeadOffice", "root", "");
                  //Assigning UPDATE_QUERY to pstmt construct, an update querry directed to
Head office site
            PreparedStatement pstmt2 =connection2.prepareStatement(UPDATE_QUERY);
```

```
//The for--loop, its sub-transactions acting as Participants

for(int index = 1; index <= 5; index++) {

//Sub-transaction-1: Inserting data into Table bankcustomers, residing in Database NairobiBranch

pstmt.setInt(1,index);//Inserting data into first_column, CustomerID

pstmt.setString(2, "Customer_" + index);//Inserting data into Second_column, CustomerName

pstmt.setString(3, "" + (4 + index));//Inserting data into Third_column, Address

pstmt.setString(4, "Nairobi");//Inserting data into Fourth_column, City

pstmt.setString(5, "25000");//Inserting data into Fifth_column, AccountBalance

pstmt.executeUpdate();// Executing the INSERT_QUERY

pstmt.close();//Terminating Sub-Transaction-1

connection.close();//Terminating database connection for Sub-Transaction-1

//Status of the voting in Database located at site, NairobiBranch

System.out.println("-----------------------------------------------------------------------------------------------");

System.out.println("NAIROBI_BRANCH_VOTE :TRANSACTION_COMMIT");


//Sub-transaction-2: Inserting data into Table bankcustomers, residing in Database KisiiBranch

pstmt1.setInt(1,index);//Inserting data into first_column, CustomerID

pstmt1.setString(2, "Customer_" + index);//Inserting data into Second_column, CustomerName

pstmt1.setString(3, "" + (4 + index));//Inserting data into Third_column, Address

pstmt1.setString(4, "Nairobi");//Inserting data into Fourth_column, City

pstmt1.setString(5, "25000");//Inserting data into Fifth_column, AccountBalance

pstmt1.executeUpdate();// Executing the INSERT_QUERY1 pstmt1.close();//Terminating Sub-Transaction-2
```

```
connection1.close();  //Terminating database connection for Sub-Transaction-2

                        //Status of the voting in Database located at site, KisiiBranch

                //          System.out.println("----------------------------------------------------------------
---------------------------------");

 System.out.println("KISII_BRANCH_VOTE :TRANSACTION_COMMIT");

     //Sub-transaction-3: Update data in Table bankcustomers, residing in Database HeadOffice

            pstmt2.executeUpdate();// Executing the UPDATE_QUERY

             pstmt2.close();//Terminating Sub-Transaction-3

             connection2.close(); //Terminating database connection for Sub-Transaction-3

                    //Status of the voting in Database located at site, HeadOffice

                        System.out.println("----------------------------------------------------------------
---------------------------------");

           System.out.println("HEAD_PFFICE_VOTE :TRANSACTION_COMMIT");

                       System.out.println("----------------------------------------------------------------
-------------------------------");

               //Status of the Coordinator
DecisionSystem.out.println("COORDINATOR_DECISION :GLOBAL_COMMIT");

                       System.out.println("----------------------------------------------------------------
-------------------------------");

         btm.commit();//Transaction Manager Commits the Group of 3-Sub-Transactions

        btm.rollback();//Transaction Manager rollsback  the Group of 3-Sub-Transactions

                //Error_Handling if commit of the 3-Sub-Transactions is not possible

           }catch (Exception ex) {

                        //ex.printStackTrace();

                        btm.shutdown();//Transaction Manager is ShutDown

                                System.out.println("----------------------------------------------------
-----------------------------------------------");}}}
```

Appendix II: Coordinator Failure-Distributed Transactions And Distributed Data Resource

Feb 15, 2015 8:01:13 PM bitronix.tm.BitronixTransactionManager logVersion

INFO: Bitronix Transaction Manager version 2.1.0

Feb 15, 2015 8:01:14 PM bitronix.tm.Configuration buildServerIdArray

WARNING: cannot get this JVM unique ID. Make sure it is configured and you only use ASCII characters. Will use IP address instead (unsafe for production usage!).

Feb 15, 2015 8:01:14 PM bitronix.tm.Configuration buildServerIdArray

INFO: JVM unique ID: <127.0.0.1>

Feb 15, 2015 8:01:14 PM bitronix.tm.recovery.Recoverer recoverAllResources

WARNING: error running recovery on resource 'TwoPCCoordinatoFailureClass1', resource marked as failed (background recoverer will retry recovery)

bitronix.tm.recovery.RecoveryException: cannot start recovery on a PoolingDataSource containing an XAPool of resource TwoPCCoordinatoFailureClass1 with 0 connection(s) (0 still available)

at bitronix.tm.resource.jdbc.PoolingDataSource.startRecovery(PoolingDataSource.java:227)

at bitronix.tm.recovery.Recoverer.recover(Recoverer.java:253)

at bitronix.tm.recovery.Recoverer.recoverAllResources(Recoverer.java:223)

at bitronix.tm.recovery.Recoverer.run(Recoverer.java:138)

at bitronix.tm.BitronixTransactionManager.<init>(BitronixTransactionManager.java:64)

at bitronix.tm.TransactionManagerServices.getTransactionManager(TransactionManagerServices.java:62)

```
    at TwoPCProtocol.TwoPCCoordinatorFailure1.main(TwoPCCoordinatorFailure1.java:31)
Caused by: java.sql.SQLException: Access denied for user "@'localhost' (using password: NO)
    at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:1055)
    at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:956)
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3558)
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3490)
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:919)
    at com.mysql.jdbc.MysqlIO.secureAuth411(MysqlIO.java:3996)
    at com.mysql.jdbc.MysqlIO.doHandshake(MysqlIO.java:1284)
    at com.mysql.jdbc.ConnectionImpl.createNewIO(ConnectionImpl.java:2142)
    at com.mysql.jdbc.ConnectionImpl.<init>(ConnectionImpl.java:781)
    at com.mysql.jdbc.JDBC4Connection.<init>(JDBC4Connection.java:46)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at
sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:3
9)
    at
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorIm
pl.java:27)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
    at com.mysql.jdbc.Util.handleNewInstance(Util.java:406)
    at com.mysql.jdbc.ConnectionImpl.getInstance(ConnectionImpl.java:352)
```

at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:284)

at com.mysql.jdbc.jdbc2.optional.MysqlDataSource.getConnection(MysqlDataSource.java:439)

at com.mysql.jdbc.jdbc2.optional.MysqlDataSource.getConnection(MysqlDataSource.java:137)

at com.mysql.jdbc.jdbc2.optional.MysqlDataSource.getConnection(MysqlDataSource.java:107)

at com.mysql.jdbc.jdbc2.optional.MysqlXADataSource.getXAConnection(MysqlXADataSource.java:47)

at bitronix.tm.resource.jdbc.PoolingDataSource.createPooledConnection(PoolingDataSource.java:274)

at bitronix.tm.resource.common.XAPool.createPooledObject(XAPool.java:283)

at bitronix.tm.resource.common.XAPool.grow(XAPool.java:400)

at bitronix.tm.resource.common.XAPool.getInPool(XAPool.java:379)

at bitronix.tm.resource.common.XAPool.getConnectionHandle(XAPool.java:123)

at bitronix.tm.resource.jdbc.PoolingDataSource.startRecovery(PoolingDataSource.java:223)

... 6 more

Feb 15, 2015 8:01:14 PM bitronix.tm.recovery.Recoverer run

INFO: recovery committed 0 dangling transaction(s) and rolled back 0 aborted transaction(s) on 0 resource(s) [] (restricted to serverId '127.0.0.1')

java.sql.SQLException: unable to get a connection from pool of a PoolingDataSource containing an XAPool of resource TwoPCCoordinatoFailureClass1 with 0 connection(s) (0 still available) -failed-

at bitronix.tm.resource.jdbc.PoolingDataSource.getConnection(PoolingDataSource.java:201)

at bitronix.tm.resource.jdbc.PoolingDataSource.getConnection(PoolingDataSource.java:207)

at TwoPCProtocol.TwoPCCoordinatorFailure1.main(TwoPCCoordinatorFailure1.java:35)

Caused by: bitronix.tm.internal.BitronixRuntimeException: incremental recovery failed when trying to acquire a connection from failed resource 'TwoPCCoordinatoFailureClass1'

at bitronix.tm.resource.common.XAPool.getConnectionHandle(XAPool.java:103)

at bitronix.tm.resource.common.XAPool.getConnectionHandle(XAPool.java:91)

At bitronix.tm.resource.jdbc.PoolingDataSource.getConnection(PoolingDataSource.java:197)
... 2 more

Caused by: bitronix.tm.recovery.RecoveryException: cannot start recovery on a PoolingDataSource containing an XAPool of resource TwoPCCoordinatoFailureClass1 with 0 connection(s) (0 still available)

at bitronix.tm.resource.jdbc.PoolingDataSource.startRecovery(PoolingDataSource.java:227)

at bitronix.tm.recovery.IncrementalRecoverer.recover(IncrementalRecoverer.java:62)

at bitronix.tm.resource.common.XAPool.getConnectionHandle(XAPool.java:100)

... 4 more

Caused by: java.sql.SQLException: Access denied for user "@'localhost' (using password: NO)

at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:1055)

at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:956)

at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3558)

at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3490)

at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:919)

at com.mysql.jdbc.MysqlIO.secureAuth411(MysqlIO.java:3996)

at com.mysql.jdbc.MysqlIO.doHandshake(MysqlIO.java:1284)

at com.mysql.jdbc.ConnectionImpl.createNewIO(ConnectionImpl.java:2142)

at com.mysql.jdbc.ConnectionImpl.<init>(ConnectionImpl.java:781)

at com.mysql.jdbc.JDBC4Connection.<init>(JDBC4Connection.java:46)

at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)

at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:39)

at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:27)

at java.lang.reflect.Constructor.newInstance(Constructor.java:513)

at com.mysql.jdbc.Util.handleNewInstance(Util.java:406)

at com.mysql.jdbc.ConnectionImpl.getInstance(ConnectionImpl.java:352)

at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:284)

at com.mysql.jdbc.jdbc2.optional.MysqlDataSource.getConnection(MysqlDataSource.java:439)

at com.mysql.jdbc.jdbc2.optional.MysqlDataSource.getConnection(MysqlDataSource.java:137)

at com.mysql.jdbc.jdbc2.optional.MysqlDataSource.getConnection(MysqlDataSource.java:107)

at com.mysql.jdbc.jdbc2.optional.MysqlXADataSource.getXAConnection(MysqlXADataSource.java:47)

at bitronix.tm.resource.jdbc.PoolingDataSource.createPooledConnection(PoolingDataSource.java:274)

at bitronix.tm.resource.common.XAPool.createPooledObject(XAPool.java:283)

at bitronix.tm.resource.common.XAPool.grow(XAPool.java:400)

at bitronix.tm.resource.common.XAPool.getInPool(XAPool.java:379)

at bitronix.tm.resource.common.XAPool.getConnectionHandle(XAPool.java:123)

at bitronix.tm.resource.jdbc.PoolingDataSource.startRecovery(PoolingDataSource.java:223)

... 6 more

java.sql.SQLException: unable to get a connection from pool of a PoolingDataSource containing an XAPool of resource TwoPCCoordinatoFailureClass1 with 0 connection(s) (0 still available) - failed-

at bitronix.tm.resource.jdbc.PoolingDataSource.getConnection(PoolingDataSource.java:201)

at bitronix.tm.resource.jdbc.PoolingDataSource.getConnection(PoolingDataSource.java:207)

at TwoPCProtocol.TwoPCCoordinatorFailure1.main(TwoPCCoordinatorFailure1.java:62)

Caused by: bitronix.tm.internal.BitronixRuntimeException: incremental recovery failed when trying to acquire a connection from failed resource 'TwoPCCoordinatoFailureClass1'

at bitronix.tm.resource.common.XAPool.getConnectionHandle(XAPool.java:103)

at bitronix.tm.resource.common.XAPool.getConnectionHandle(XAPool.java:91)

at bitronix.tm.resource.jdbc.PoolingDataSource.getConnection(PoolingDataSource.java:197)

... 2 more

Caused by: bitronix.tm.recovery.RecoveryException: cannot start recovery on a PoolingDataSource containing an XAPool of resource TwoPCCoordinatoFailureClass1 with 0 connection(s) (0 still available)

at bitronix.tm.resource.jdbc.PoolingDataSource.startRecovery(PoolingDataSource.java:227)

at bitronix.tm.recovery.IncrementalRecoverer.recover(IncrementalRecoverer.java:62)

at bitronix.tm.resource.common.XAPool.getConnectionHandle(XAPool.java:100)

... 4 more

Caused by: java.sql.SQLException: Access denied for user ''@'localhost' (using password: NO)

at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:1055)

at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:956)

at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3558)

at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3490)

at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:919)

at com.mysql.jdbc.MysqlIO.secureAuth411(MysqlIO.java:3996)

at com.mysql.jdbc.MysqlIO.doHandshake(MysqlIO.java:1284)

at com.mysql.jdbc.ConnectionImpl.createNewIO(ConnectionImpl.java:2142)

at com.mysql.jdbc.ConnectionImpl.<init>(ConnectionImpl.java:781)

at com.mysql.jdbc.JDBC4Connection.<init>(JDBC4Connection.java:46)

at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)

at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:39)

at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:27)

at java.lang.reflect.Constructor.newInstance(Constructor.java:513)

at com.mysql.jdbc.Util.handleNewInstance(Util.java:406)

at com.mysql.jdbc.ConnectionImpl.getInstance(ConnectionImpl.java:352)

at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:284)

at com.mysql.jdbc.jdbc2.optional.MysqlDataSource.getConnection(MysqlDataSource.java:439)

at com.mysql.jdbc.jdbc2.optional.MysqlDataSource.getConnection(MysqlDataSource.java:137)

at com.mysql.jdbc.jdbc2.optional.MysqlDataSource.getConnection(MysqlDataSource.java:107)

at com.mysql.jdbc.jdbc2.optional.MysqlXADataSource.getXAConnection(MysqlXADataSource.java:47)

at bitronix.tm.resource.jdbc.PoolingDataSource.createPooledConnection(PoolingDataSource.java:274)

at bitronix.tm.resource.common.XAPool.createPooledObject(XAPool.java:283)

at bitronix.tm.resource.common.XAPool.grow(XAPool.java:400)

at bitronix.tm.resource.common.XAPool.getInPool(XAPool.java:379)

at bitronix.tm.resource.common.XAPool.getConnectionHandle(XAPool.java:123)

at bitronix.tm.resource.jdbc.PoolingDataSource.startRecovery(PoolingDataSource.java:223)

... 6 more

Feb 15, 2015 8:01:14 PM bitronix.tm.BitronixTransactionManager shutdown

INFO: shutting down Bitronix Transaction Manager

Process completed.

Appendix III: Two Phase Commit Protocol With Transaction Clustering

```
package TwoPCCoordinator;

// Author: Teresa Abuya, Msc (Computer Systems), Jkuat

//Java imports for database connectivity

        import java.sql.DriverManager;

        import java.sql.Connection;

        import java.sql.PreparedStatement;

        import java.sql.SQLException;

        import java.text.DateFormat;

        import java.text.SimpleDateFormat;

        import java.sql.Statement;

        import java.sql.Connection;

        import java.sql.PreparedStatement;

        import java.sql.ResultSet;

                //Bitronix Transaction manager imports for database transactions monitoring
and control

                        import bitronix.tm.BitronixTransactionManager;

                import bitronix.tm.TransactionManagerServices;

                import bitronix.tm.resource.jdbc.PoolingDataSource;

//end of imports

                 //Declaration of the class, "Coordinator
```

```
public class Coordinator {

//Constant constructs declaration and instantiation

        private static final String DATABASE="KisiiBranch";

        private static final String DB_DRIVER = "com.mysql.jdbc.Driver";

        private static final String DB_CONNECTION =
"jdbc:mysql://localhost:3306/customerdetails";

        private static final String DB_USER = "root";

        private static final String DB_PASSWORD = "";

        private static final String USER_NAME="Teresa";

        private static final String PASSWORD="Abuya";

    //Database querries instantiation

            private static final String INSERT_QUERY="insert into
Bankcustomers(CustomerID,CustomerName,Address,City,AccountBalance)values (?,?,?,?,?)";

            private static final String INSERT_QUERY1="insert into
bankcustomers(CustomerID,CustomerName,Address,City,AccountBalance)values (?,?,?,?,?)";

            private static final String UPDATE_QUERY="Update
bankcustomers SET AccountBalance='25000'";

        //      Beginning of main class

    public static void main(String[] argv) {

    //Setting of the database parameters

            PoolingDataSource mySQLDS=new PoolingDataSource();
    MySQLDS.setClassName("com.mysql.jdbc.jdbc2.optional.MysqlXADataSource");

                mySQLDS.setUniqueName("Coordinator");
```

```
                mySQLDS.setMaxPoolSize(3);

                mySQLDS.getDriverProperties().setProperty("databaseName", DATABASE);
//Bitronix transaction manager initialization

                BitronixTransactionManager btm
=TransactionManagerServices.getTransactionManager();
 //Transaction Clustering for Coordinator and site failure circumvention

                try {

                //Starting Bitronix transaction manager

          btm.begin();

          //Three Distributed databases are assumed, residing in Nairobi, Kisii and Head Office
 // Initializing JDBC Drivers and Database site NairobiBranch, with user as root

                Connection connection = DriverManager.getConnection(

              "jdbc:mysql://localhost:3306/NairobiBranch", "root", "");

        //Assigning INSERT_QUERRY to pstmt construct, an insert querry directed to Nairobi site

          PreparedStatement pstmt =connection.prepareStatement(INSERT_QUERY);
//Initializing JDBC Drivers and Database site KisiiBranch, with user as root

          Connection connection1 = DriverManager.getConnection(

              "jdbc:mysql://localhost:3306/KisiiBranch", "root", "");

        // Assigning INSERT_QUERRY1 to pstmt construct, an insert querry directed to Kisii site

          PreparedStatement pstmt1 =connection1.prepareStatement(INSERT_QUERY1);

         // Initializing JDBC Drivers and Database site KisiiBranch, with user as root

          Connection connection2 = DriverManager.getConnection(
```

```
                "jdbc:mysql://localhost:3306/HeadOffice", "root", "");

    //Assigning UPDATE_QUERY to pstmt construct, an update querry directed to Head office
site

        PreparedStatement pstmt2 =connection2.prepareStatement(UPDATE_QUERY);

        //The for--loop, its sub-transactions acting as Participants

      for(int index = 1; index <= 5; index++) {

//Sub-transaction-1: Inserting data into Table bankcustomers, residing in Database
NairobiBranch

            pstmt.setInt(1,index);//Inserting data into first_column, CustomerID

pstmt.setString(2, "Customer_" + index);//Inserting data into Second_column, CustomerName

                    pstmt.setString(3, "" + (4 + index));//Inserting data into Third_column, Address

                    pstmt.setString(4, "Nairobi");//Inserting data into Fourth_column, City

                    pstmt.setString(5, "25000");//Inserting data into Fifth_column, AccountBalance

              pstmt.executeUpdate();// Executing the INSERT_QUERY

              pstmt.close();//Terminating Sub-Transaction-1

              connection.close();//Terminating database connection for Sub-Transaction-1

      //Status of the voting in Database located at site, NairobiBranch

              System.out.println("----------------------------------------------------------------------------
----------");

                        System.out.println("NAIROBI_BRANCH_VOTE
:TRANSACTION_COMMIT");
```

```
                    //


    //Sub-transaction-2: Inserting data into Table bankcustomers, residing in Database
KisiiBranch

                        //

                            pstmt1.setInt(1,index);//Inserting data into first_column,
CustomerID

                            pstmt1.setString(2, "Customer_" + index);//Inserting data into
Second_column, CustomerName

                        pstmt1.setString(3, "" + (4 + index));//Inserting data into
Third_column, Address

                            pstmt1.setString(4, "Nairobi");//Inserting data into
Fourth_column, City

                            pstmt1.setString(5, "25000");//Inserting data into Fifth_column,
AccountBalance

        pstmt1.executeUpdate();// Executing the INSERT_QUERY1

        pstmt1.close();//Terminating Sub-Transaction-2

        connection1.close();  //Terminating database connection for Sub-Transaction-2

                //

      //Status of the voting in Database located at site, KisiiBranch

            //

                System.out.println("----------------------------------------------------------------
--------------------------------");
```

```
                    System.out.println("KISII_BRANCH_VOTE :TRANSACTION_COMMIT");

                    //

    //Sub-transaction-3: Update data in Table bankcustomers, residing in Database HeadOffice
            //
        pstmt2.executeUpdate();// Executing the UPDATE_QUERY
          pstmt2.close();//Terminating Sub-Transaction-3
          connection2.close(); //Terminating database connection for Sub-Transaction-3
                                //Status of the voting in Database located at site, HeadOffice
System.out.println("HEAD_PFFICE_VOTE :TRANSACTION_COMMIT");
  //Status of the Coordinator Decision
                    System.out.println("COORDINATOR_DECISION :GLOBAL_COMMIT");
        btm.commit();//Transaction Manager Commits the Group of 3-Sub-Transactions
 btm.rollback();//Transaction Manager rollsback  the Group of 3-Sub-Transactions{
          //Error_Handling if commit of the 3-Sub-Transactions is not possible
                        }catch (Exception ex){
                                    //ex.printStackTrace();
                btm.shutdown();//Transaction Manager is ShutDown
                                System.out.println("--------------------------------------------------
------------------------------------------------")}}}}
```