Volume 5, No. 12, December 2014
Journal of Global Research in Computer Science
ISSN-2229-371X
RESEARCH PAPER
Available Online at www.jgrcs.info

AN IMPROVED FAILURE RECOVERY ALGORITHM IN TWO-PHASE COMMIT PROTOCOL FOR TRANSACTION ATOMICITY

Teresa K. Abuya*
Computer Science, Kisii University,
Kenya
tkwambokaa@gmail.com

Dr.Richard M. Rimiru, PhD
Computer Science, Jomo Kenyatta
University of agriculture & Technology
Kenya
rimirurm@gmail.com

Dr.Cheruiyot W.K, PhD
omputer Science, Jomo Kenyatta
University of agriculture & Technology
Kenya
wilchery68@gmail.com

Abstract: Distributed database systems poses different problems when accessing distributed and replicated databases. Two-Phase Commit protocol (2PC) is a standard algorithm for safeguarding the ACID properties of transactions in distributed systems. It ensures that every single transaction in a distributed system is executed to its completion or one of its operations is committed. In a distributed database system, a transaction blocks during Two-Phase Commit (2PC) processing if the coordinator site fails and at the same time some participant site has declared itself ready to commit the transaction. The blocking phenomena reduce the availability of the system since the participants keep locks on resources until they receive the next message from the coordinator after its recovery. A backup coordinator was employed to deal with the problem but it introduced more communication overheads. This paper addresses the problem of transaction blocking while reducing communication overheads using a simulation algorithm. The simulation algorithm was developed using Jcreator with mySQL acting as a back end data manager, the Bitronix transaction manager (BTM) which is a simple but complete implementation of Java applications. Bitronix transaction manger is a perfect choice for a project using transaction capabilities by sing Java Transfer Manager (JTM) facade. The simulation transaction algorithm will minimize coordinator failure problem in distributed transactions.

1.0 INTRODUCTION

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. Concurrency control has proofed to be very difficult task for distributed systems because of absence of global clock and lack of shared memory. 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. Distributed systems allow people to query the system from anywhere on the distributed network and it's not necessary to know where on the network the data being sought is located. The user issues a query and then result is returned, a feature known as location transparency [2]

A distributed database combines two different technologies used for data processing: database systems and computer networks. Retrieval of data from different sites is called query processing. Query processing is more complex and difficult in distributed environment thus the goal of a Distributed Query Processor is to execute such queries efficiently in order to minimize the response time and total communication cost associated with a query database. 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 [4]. 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 participant waits for a final answer from the coordinator, if he should commit the transaction or not, the participant is blocked as long as the coordinator is down. This is a problem associated

with Two-Phase Commit (2PC) protocol.To enhance its performance, a simulation algorithm in 2PC is created to show how coordinator failure is minimized in distributed transactions [10]

2.0 ACID properties of transactions

The concept of a database transaction or atomic transaction has evolved in order to enable both a well understood database system behaviour 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[5]. 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. Every database transaction obeys ACID (Atomicity, Consistency, Isolation, and Durability) property rules in a database system: [12]

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.

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. Providing isolation is the main goal of concurrency control.

Durability: This guarantees that the effects of successful committed transactions must persist through crashes typically by recording the transaction's effects and its commit event in a non-volatile memory.

3.0 Two-Phase Commit Protocol Analysis

The Two-Phase Commit Protocol (2PC) is a distributed algorithm used in computer networks and distributed database systems. 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. 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 [13]

3.1. Atomicity in 2PC

Atomicity is ensured when either all the operations associated with a program unit are executed to completion or none are performed. Ensuring atomicity in a distributed system requires a transaction coordinator which is responsible for the following:-

- Starting the execution of the transaction.
- Breaking the transaction into a number of sub-transactions, and distributing these sub-transactions to the appropriate sites for execution.
- Coordinating the termination of the transaction, this may result in the transaction being committed at all sites or aborted at all sites.

The execution of 2PC is initiated by the coordinator after the last step of the transaction has been reached.

When the protocol is initiated, the transaction may still be executing at some of the local sites. The protocol involves all the local sites at which the transaction executed. Let *T* be a transaction initiated at site *Si* and let the transaction coordinator at *Si* be *Ci*

Phase 1 of 2PC is usually called "Prepare phase". The following are the actions performed during this phase:

- *Ci* adds <prepare *T*> record to the log.
- *Ci* sends <prepare *T*> message to all sites.

When a site receives a <prepare T> message, the transaction manager determines if it can commit the transaction.

- If no: Add <no *T*> record to the log and respond to
- *Ci* with <abort *T*>.
- If yes: Add <ready *T*> record to the log, force all log records for *T* onto stable storage and transaction manager sends <ready *T*> message to *Ci.*

The Coordinator collects responses from all sites. If all respond "ready", the final decision is commit. If at least one response is "abort", the final decision is abort. If at least one participant fails to respond within a time out period, the final decision is abort.

Phase 2 of 2PC is usually called "Committing to *T* in the database". The following are the actions performed during this phase:

- The coordinator adds a decision record <abort *T*> or <commit *T*> to its log and forces the record onto stable storage.
- Once that record reaches stable storage it is irrevocable even if failures occur.
- The coordinator sends a message to each participant informing it of the decision commit or abort.
- Participants take appropriate action locally.

Site failure in 2PC is handled in the following manner

1. If the log contains a <commit *T*> record, the site executes redo *(T).*
2. If the log contains an <abort *T*> record, the site executes undo *(T).*
3. If the log contains a <ready *T*> record, consult *Ci*. If *Ci* is down, site sends query-status *T* message to the other sites.

4. If the log contains no control records concerning T, the site executes undo *(T).*
Coordinator failure in 2PC is handled in the following manner:
1. If an active site contains a <commit *T*> record in its log, the *T* must be committed.
2. If an active site contains an <abort *T*> record in its log, then *T* must be aborted.
3. If some active site does not contain the record <ready *T*> in its log then the failed coordinator *Ci* cannot have decided to commit *T.* Rather than wait for *Ci* to recover, it is preferable to abort *T* .
4. If all active sites have a <ready *T*> record in their logs, but no additional control records, there is a need to wait for the coordinator to recover.
5. Blocking problem: *T* is blocked pending the recovery of site *Si.*

3.2. 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 Query processing flow

A query process flow shows how transactions are committed in a distributed environment using Two-Phase Commit protocol [11]

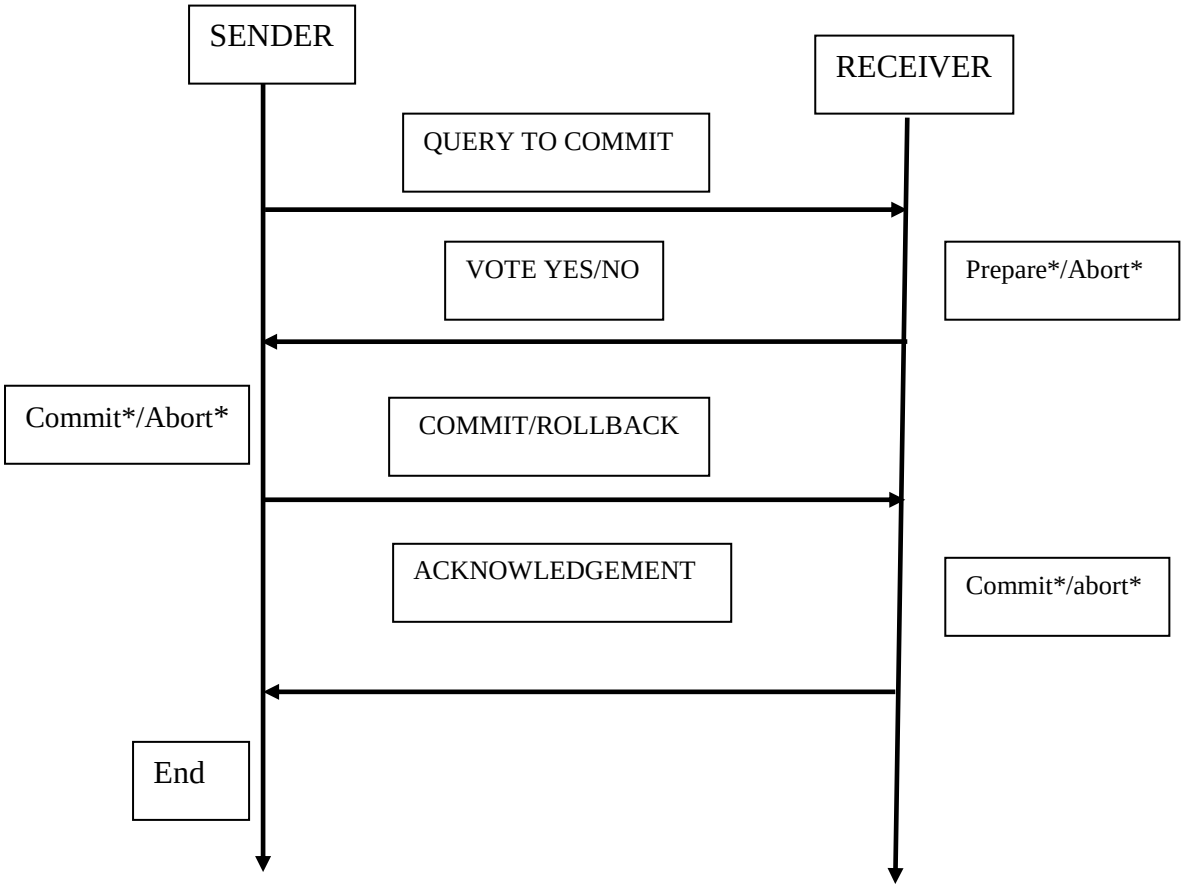

Fig 1.0 Query processing flow

From the above figure we can explain the success and failure of distributed transactions in 2PC as follows;

Success: If the sender received an agreement message from all receivers during the commit-request phase:
a) The sender sends a commit message to all the receivers.
b) Each receiver completes the operation, and releases all the locks and resources held during the transaction.
c) Each receiver sends an acknowledgment to the sender.
d) The sender completes the transaction when all acknowledgments have been received.

Failure: If any receiver votes No during the commit-request phase or the sender's timeout expires:
a) The sender sends a rollback message to all the receivers.

b) Each receiver undoes the transaction using the undo log, and releases the resources and locks held during the transaction.
c) Each receiver sends an acknowledgement to the sender.
d) The sender undoes the transaction when all acknowledgements have been received.

3.4 Blocking problem in 2PC

In 2PC protocol, 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 Blocking problem is described with the given circumstances that, if the coordinator fails to operate and at the same time some participants has confirmed itself to commit state. In this circumstance to end the blocked transaction, the participant should wait until the coordinator gets recovery. The blocked transactions continue to keep all the resources until they obtain the final decision from the coordinator after its recovery [8]

3.5 A backup coordinator in 2PC flowchart

A backup coordinator was proposed to address coordinator failure and blocking problem in 2PC which worked in this way: It works in the following way:- A primary coordinator generates a prepare message for a 2PC distributed transaction, including an address of a backup coordinator. The primary coordinator maintains a transaction log of the distributed transaction, wherein the transaction log is accessible to both the primary coordinator and the backup coordinator. The prepare message is sent to multiple participants. The primary coordinator fails over to the backup coordinator without interrupting the distributed transaction. The diagram below gives a general flow of information in a distributed environment where a backup coordinator is used in a 2PC[12].

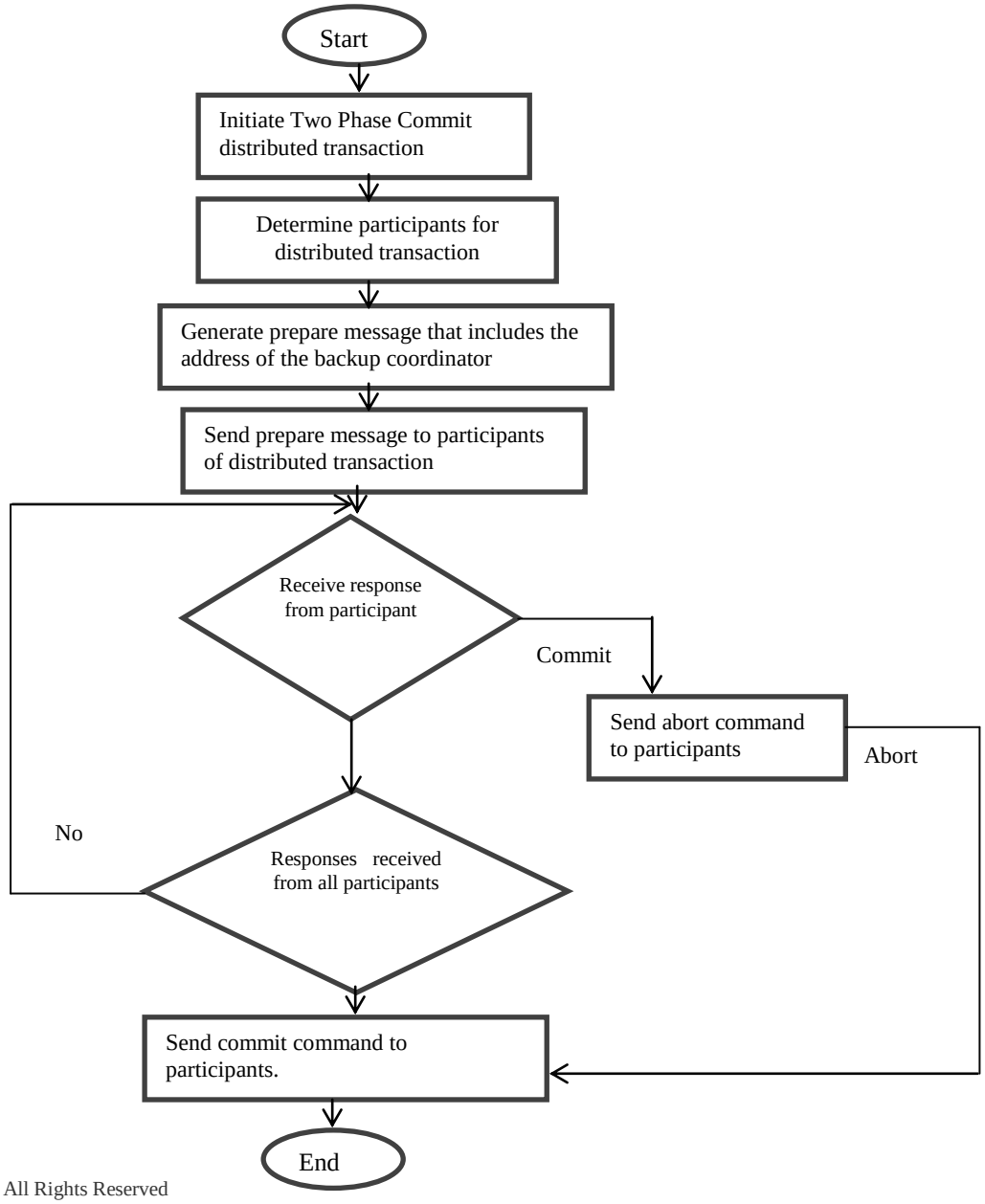

Figure 1.1 Use of backup coordinator in a 2PC protocol flowchart

4.0 Coordinator failure in Two-Phase Commit Protocol

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[7]
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 [9].

4.1. Tools and Specifications for the transaction algorithm

In order to show 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 virtual machine environment

5.0 2PC Coordinator Failure Simulation

Two practical implementations of coordinator failures were carried out to demonstrate the fact that 2PC is blocking. Bitronix transaction manager, *mySQL* server and *JCreator* IDE were used to simulate a two-phase commit protocol failure. It 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 that were namely, the *TwoPCCoordinatorFailureClass* and the and *TwoPCCoordinatorFailureClass1*. The first main class, which is the, *TwoPCCoordinatorFailureClass* was to show the coordinator failure for distributed transactions involving one database while the second class which is the, *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 the code below, *"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, 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, the code,

```
"catch (Exception ex) {
        ex.printStackTrace();
try {
                        btm.rollback();
                } catch (Exception e) {
                        e.printStackTrace();
                }"
```

This shows a transaction branch that results when the transaction class cannot commit a transaction.

5.1 Relationship among distributed system entities

The relationships among these entities are shown Figure 1.2 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.

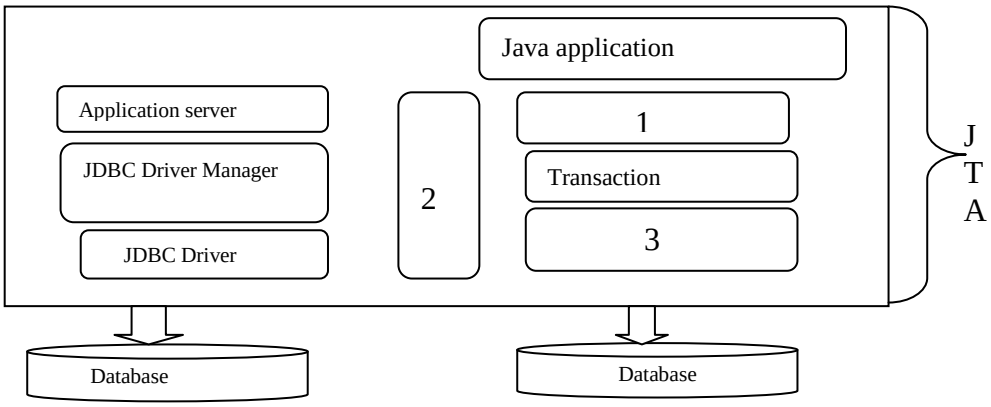

Figure 1.2: Relationship Among Distributed System Entities (Abuya et.al,2015)

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*.

5.2. Coordinator Failure Demonstration in 2PC

To demonstrate coordinator failure, the data resource was put online and ran the source codes of coordinator failure involving distributed transactions directed towards a distributed data resource. Table 1.1 below gives a snippet of the output of the Bitronix transaction manager log file where there is a coordinator failure in distributed transactions and distributed data resource.

Feb 5, 2015 7:12:52 AM bitronix.tm.recovery.Recoverer recoverAllResources
host 127.0.0.1
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)
atbitronix.tm.resource.jdbc.PoolingDataSource.startRecovery(PoolingDataSource.java:227)
at bitronix.tm.recovery.Recoverer.recover(Recoverer.java:253)
atbitronix.tm.recovery.Recoverer.recoverAllResources(Recoverer.java:223)
at bitronix.tm.recovery.Recoverer.run(Recoverer.java:138)

Table 1.1: 2PC coordinator failure

The two phase commit algorithm to insert and retrieve the data into these tables was written. 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)*'.

5.3 Coordinator Failure-Distributed-Transactions and Distributed Data Resource

As shown in the table 1.2 below, 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.


```
……
INFO: JVM unique ID: <127.0.0.1>
Feb 5, 2015 7:12:52 AM 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)
```

Table 1.2 Coordinator Failure-Distributed Transactions and Distributed Data Resource

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)*'.

6.0 Proposed transaction algorithm

To address coordinator failure and transaction blocking problem in two-phase commit protocol, a simulated transaction algorithm for optimizing distributed transactions is designed.

6.1 Architecture of the algorithm

The algorithm consists 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 t 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.

6.2 Overall Simulation Architecture

Figure 1.3 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*.

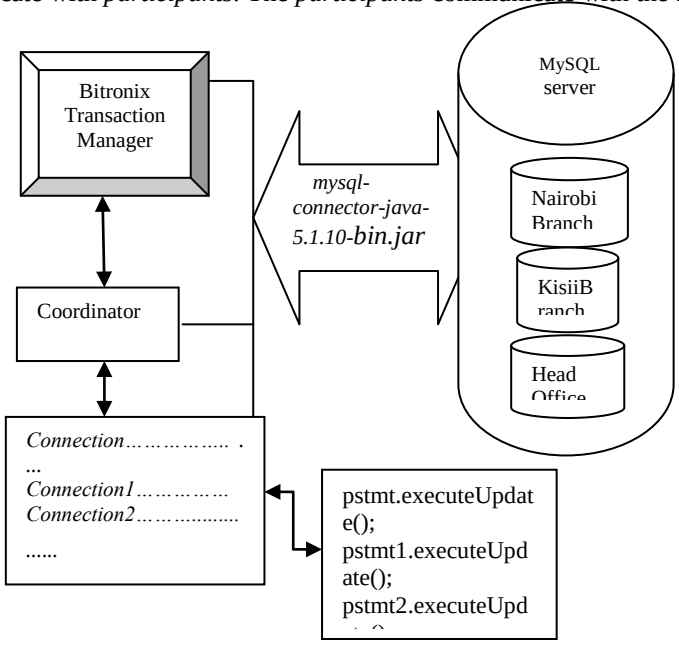

Fig 1.3 Overall simulation architecture (Abuya *et.al,* 2015)

6.3 Procedure for the simulation algorithm

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.* The structure for these tables is similar to the one in Figure 1.4 above.
4.

```
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++) {
//Sub-transaction1: 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
 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
                    //
         //Status of the voting in Database located at site, KisiiBranch
                //
                    System.out.println("-----------------------------------------------------------------------------------------------
---");
                    System.out.println("KISII_BRANCH_VOTE :TRANSACTION_COMMIT");
```

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.

7.0 Transaction algorithm output in Two-Phase Commit protocol

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

Table 1.4. Two Phase Commit output

When an algorithm was run all the sites voted TRANSACTION-COMMIT and the coordinator decision was GLOBAL-COMMIT. This demonstrated the success of a clustering algorithm in dealing with coordinator failure. Either all transactions are committed at once or none commits at all.
A check on the three sites confined that these transactions had actually committed as shown in table 1.5,1.6 and 1.7 below.

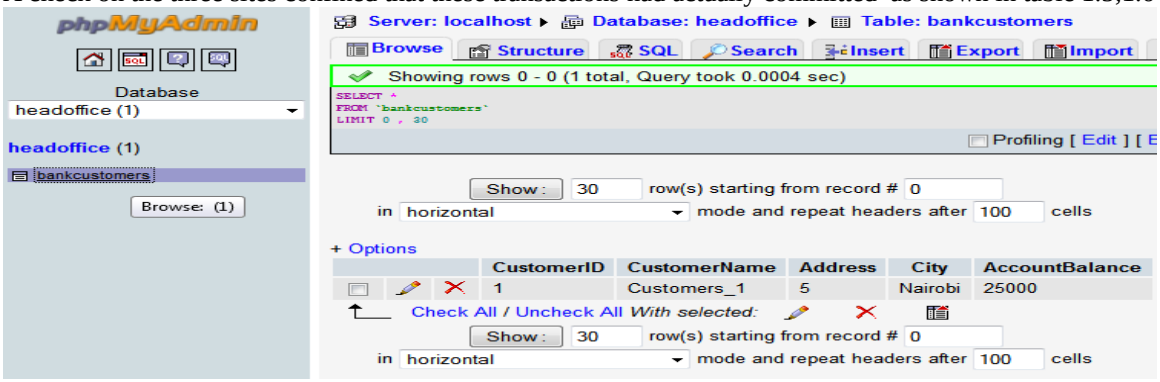

Table 1.5 headoffice site final status after commit

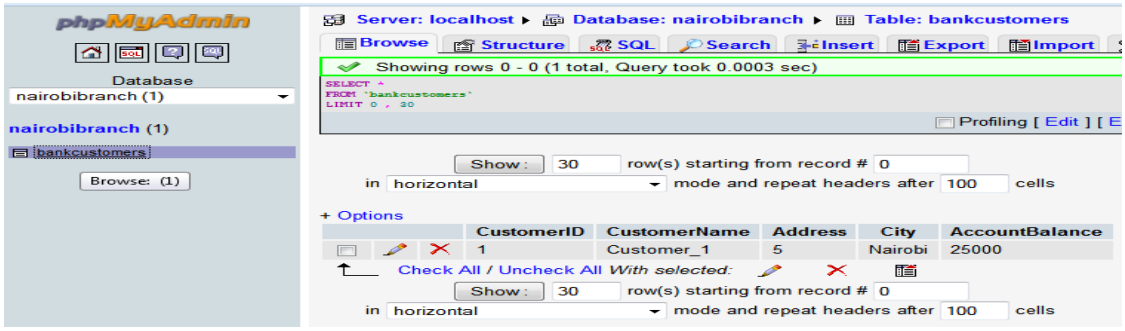

Table 1.6 Nairobi Branch site final status after commit

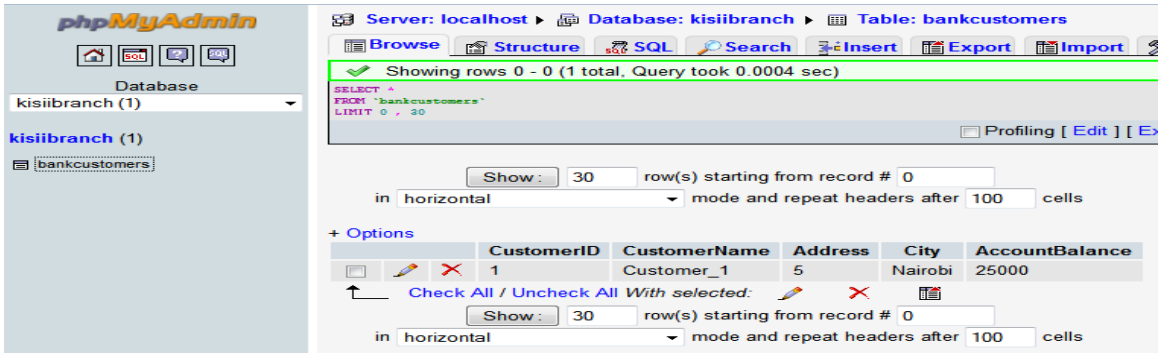

Table 1.7. .KisiiBranch site final status after commit

The above tables demonstrate concurrency and blocking controls in the new clustered algorithm. 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 claimed 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.

Therefore this transaction clustering algorithm addresses the problem of coordinator failure while ensuring that no blocking of transactions as they commit as a group or bunch. Either all or none of the transactions commit condition achieves the atomicity property of transactions.

8.0 Conclusions

The results obtained indicated that by using the proposed algorithm, coordinator failure in transactions associated with the current Two-Phase Commit can be reduced. This was achieved by eliminating transaction partitioning, which is an inherent feature of the current Two Phase Commit protocol(2PC). 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, we 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 using 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 atomic transactions. The new algorithm had an improved performance in as far as 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. Finally, there is need to implement this algorithm in other backbends, such as *SQL* and oracle servers.

REFERENCES

[1]M.Castro and B. Liskov, “Practical Byzantine Fault Tolerance and Proactive Recovery,”ACM Trans. Computer Systems,vol.20, no. 4,pp. 398-461,Nov.2002.

[2] Ashwini G. Rao:Memory Constrained DBMSs with updates, 2003

[3] Yousef J. Al-Houmaily,Panos K. Chrysanthis:1-2 PC: The one-two phase atomic commit protocol., 2004.

[4] W. Fokkink, J. F. Groote, J. Pang, B. Badban, and J. van de Pol, “Verifying a sliding window protocol in mCRL,” inAMAST, ser. Lecture Notes in Computer Science, C. Rattray, S. Maharaj, and C. Shankland, Eds., vol. 3116. Springer, 2004, pp. 148–163.

[5] Cabrera, L. F., et al. (2005): Web Services Atomic Transaction.

[6] Edgar Nett, Reaching Consensus - A Basic Problem in Cooperative Applications, 2007.

[7] O'Brien, J. & Marakas, G.M.(2008) Management Information Systems (pp. 185-189). New York, NY: McGraw-Hill Irwin

[8] Y, Amir, B.A. Coan, J.Kirsch, and J. Lane,” Byzantine Replication under Attack,” Proc. IEEE Int’l Conf. Dependable Systems and Networks, pp. 105-144, June 2008.

[9]C. Baier and J.-P. Katoen, Principles of Model Checking (Representation and Mind Series). The MIT Press, 2008.

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

[11] K. Tabassum , Taranum F, Damodaram A“*A Simulation of Performance of Commit Protocols in Distributed Environment”*, in PDCTA, CCIS 203, pp. 665–681, Springer, 2011.

[12] S. Fahd and Tarek Helmy . Al-Otaibi,” *Dynamic Load-Balancing Based on a Coordinator and Backup Automatic Election in Distributed Systems*”, International Journal of Computing & Information Sciences Vol. 9, No. 1, April 2011.

[13] PoonamSingh,ParulYadav,AmalShukla,SanchitLohia,”*An Extended Three Phase Commit Protocol for concurrency control in Distributed Systems,*” International Journal for computer applications,Vol 21.No 10 May,2011.

[14]R. Schapiro and R. Milistein, *"Failure recovery in a distributed database system,*" in Proc. 1978COMPCONConf.,Sept. 2012

AUTHORS BIOGRAPHIES

Teresa K. Abuya: Received Bsc. Information Technology from Jomo Kenyatta University of Agriculture & Technology (JKUAT),Kenya; Msc .in Computer Systems from JKUAT. Serving as an assistant lecturer at Kisii University, Kenya. Her research interests include but are not limited to: Distributed database systems, Mobile & cloud computing, Internet of Things, ICT for Development, Data mining & Knowledge discovery.

Dr.Richard M. Rimiru: Received Bsc.in Statistics & Computer Science from Jomo Kenyatta University of Agriculture & Technology(JKUAT),Kenya;Msc.in Computer Science from National University of Science & Technology, Zimbambwe; PhD in Computer Science & Technology,Central South University(CSU) ,Changsha,P.R.China.Serving as a senior lecturer in School of Computing & Information Technology at JKUAT, Kenya. His research interests include but not limited to: Database systems, artificial intelligence knowledge based systems,computer networks & mobile computing.

Dr.Cheruiyot W. Kipruto: Received Bsc. In Statistics & Computer Science from Jomo Kenyatta University of Agriculture & Technology(JKUAT),Kenya; Msc.inComputer Application Technology, CentralSouth University(CSU),P.R.China; PhD in Computer Science &Technology ,Central South University(CSU)
ChangshaP.R.China.Serving as a senior lecturer in School of Computing & Information Technology at JKUAT,Kenya. His Research interest include but not limited to: Multimedia Data Retrieval, Internet of Things,Evolutionary Computation for Optimization, Digital Image Processing and ICT for Development.

.

