Georgia State University
Digital Archive @ GSU

Computer Science Theses | Department of Computer Science

12-5-2006

An Agent Based Transaction Manager for Multidatabase Systems

Sugandhi Madiraju
smadiraju1@student.gsu.edu

Follow this and additional works at: http://digitalarchive.gsu.edu/cs_theses

Recommended Citation
Madiraju, Sugandhi, "An Agent Based Transaction Manager for Multidatabase Systems" (2006). *Computer Science Theses.* Paper 36.

This Thesis is brought to you for free and open access by the Department of Computer Science at Digital Archive @ GSU. It has been accepted for inclusion in Computer Science Theses by an authorized administrator of Digital Archive @ GSU. For more information, please contact digitalarchive@gsu.edu.

AN AGENT BASED TRANSACTION MANAGER FOR MULTIDATABASE SYSTEMS

by

SUGANDHI MADIRAJU

Under the Direction of Raj Sunderraman

ABSTRACT

A multidatabase system (MDBMS) is a facility that allows users to access data located in multiple autonomous database management systems (DBMSs) at different sites. To ensure global atomicity for multidatabase transactions, a reliable global atomic commitment protocol is a possible solution. In this protocol a centralized transaction manager (TM) receives global transactions, submits subtransactions to the appropriate sites via AGENTS. An AGENT is a component of MDBS that runs on each site; AGENTS after receiving subtransactions from the transaction manager perform the transaction and send the results back to TM. We have presented a unique proof-of-concept, a JAVA application for an Agent Based Transaction Manager that preserves global atomicity. It provides a user friendly interface through which reliable atomic commitment protocol for global transaction execution in multidatabase environment can be visualized. We demonstrated with three different test case scenarios how the protocol works. This is useful in further research in this area where atomicity of transactions can be verified for protocol correctness.

INDEX WORDS: MDBMS, Global Transaction, Transaction Manager, Atomicity, JAVA Sockets.

AN AGENT BASED TRANSACTION MANAGER FOR MULTIDATABASE SYSTEMS

by

SUGANDHI MADIRAJU

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Science

Georgia State University

2006

Copyright by
Sugandhi Madiraju
2006

AN AGENT BASED TRANSACTION MANAGER FOR MULTIDATABASE SYSTEMS

by

SUGANDHI MADIRAJU

Major Professor: Dr. Raj Sunderraman
Committee: Dr. Saeid Belkasim
Dr. Yingshu Li

Electronic Version Approved:

Office of Graduate Studies
College of Arts and Sciences
Georgia State University
December 2006

Acknowledgements

I would like to thank my advisor, Dr. Raj Sunderraman, for his encouragement, advice and guidance throughout my thesis work which made my graduate studies a wonderful experience of my life.

I would like to thank Dr. Saeid Belkasim and Dr. Yingshu Li for reviewing my manuscript and providing me fine pointers to meet the standards.

I would not have realized my goal without the love of my life- Jaanu. I would like to thank my roommates- Srilaxmi, Mugdha and my friends – Shilpa, Rohny, Phani, Nofiya, Neelima, Pranay, Sangeetha, Harsh, Piyaphol, Wiwek and others.

Finally, I would like to thank my family- Mummy, Pappu and my brother Praveen for their love and constant support which made this possible.

Dedicated to everyone who was a part of this
For all the support

Table of Contents

List of Figures…….viii

1 INTRODUCTION……..1

1.1 Local Autonomy……2

1.2 Problem Statement…….....3

2TRANSACTION MANAGEMENT……..……..5

2.1 Classification of Transactions……6

2.2 ACID Properties of Transactions……10

3 ARCHITECTURE……..……..……..12

3.1 A Multidatabase System Architecture……12

3.2 Multidatabase Transaction Manager and AGENTS……13

3.3 State Transitions of TM and AGENT……..13

4 MULTIDATABASE TRANSACTION MANAGEMENT ISSUES…….16

4.1 Concurrency Control……16

4.2 Global Serializability in Multidatabases……..17

4.3 Recovery from Failures……19

4.4 Global Deadlock Problem……20

4.5 Issues related to Global Atomicity……...21

5 SOCKET PROGRAMMING……..22

5.1 Sockets…….22

5.2 Communication between TM and AGENTS using Sockets……25

6 DESIGN & IMPLEMENTATION..……26

6.1 Functions of TM and AGENTS……...26

6.2 Commit Protocol……..27

6.3 Termination Protocol……...27

6.4 Recovery Protocol……28

7 TEST CASE SCENARIOS…….……30

7.1 Commit…….....30

7.2 Abort……35

7.3 Site Failure….....40

8 RELATED WORK……...43

9 CONCLUSION AND FUTURE WORK.....….....47

9.1 Implementation Details……47

9.2 Future Extensions……47

10 BIBLIOGRAPHY...……....49

Appendix A……...53

Appendix B……...62

List of Figures

Figure 1. Structure of a Nested Transaction……9

Figure 2. Distributed Transaction……10

Figure 3. A Multidatabase System Architecture……12

Figure 4. State Transitions of TM……13

Figure5. State Transitions of AGENTS……...14

Figure 6. Concurrency control by TM for Distributed Transactions……17

Figure 7. Client-Server Communication using Sockets……...22

1. INTRODUCTION

Progress in communication and database technologies has changed the user data processing environment. The present data processing environment can be described by applications that require processing to be performed at multiple sites, also need to access data located at multiple sites of a network. A multidatabase system (MDBS) is software that runs on top of the individual database management systems (DBMSs) which is maintained at each site. Transactions in MDBS may be either local or global. Local transactions are directly submitted to individual local DBMS (LDBMS) and they access only local data. A global transaction consists of one or more subtransactions which may access data from more than one LDBMS. Local transaction and subtransaction execution is controlled by LDBMS whereas global transaction execution is controlled by MDBMS. To ensure natural integration of several LDBMSs, MDBMS should preserve its local autonomy [1].Transaction management; one of the many issues in MDBS design raises problems like concurrency control, commitment, and recovery [2]. These challenges have been met earlier with an assumption that failures don't occur [3, 4, 5, 6, and 7]. In a failure free environment strict two-phase locking assures global atomicity. However that cannot be applied incase of a subtransaction failure [8]. Hence a global atomic commitment protocol that can handle failures is necessary. [9] Proposed a global atomic commitment protocol in presence of failures but at the cost of violation of local autonomy. The approach taken in [10] achieves global atomicity and can recover from failures. In this protocol a centralized transaction manager (TM) receives global transactions, submits subtransactions to the appropriate sites via agents. AGENT is a

component of MDBS that runs on each site; AGENTS after receiving subtransactions from the transaction manager perform the transaction and send the results back to TM. We have presented a unique proof-of-concept, a JAVA API for an agent based transaction manager that preserves global atomicity. It provides a user friendly interface through which reliable atomic commitment protocol for global transaction execution in multidatabase environment can be visualized. . We demonstrated with three different test case scenarios how the protocol works. This helps in further research in this area where atomicity of transactions can be verified for protocol correctness.

1.1 Local Autonomy

MDBMS considers each LDBMS as a black box that operates autonomously, without the knowledge of either other LDBMS or the MDBMS. Local autonomy is the key feature that distinguishes MDBMS from conventional distributed database systems. There are 3 main types of autonomy [11, 1].

1. Design autonomy

Changes cannot be made to the LDBMS to accommodate with MDBMS. Changing the existing DBMS is expensive, may result in performance degradation and may make pre-existing applications inoperative.

Execution autonomy

Each LDBMS should have complete control over the execution of transactions at its site. This implies that a LDBMS can abort a transaction executing at its site at any point of execution.

Communication autonomy

LDBMS integrated by the MDBMS are not able to coordinate the actions of global transactions executing at several sites. This constraint implies that LDBMS does not share their control information with other LDBMS or with the MDBS.

Different DBMS may have different autonomy levels. For example, some sites may be willing to coordinate (low communication autonomy) a global transaction while others may not (high communication autonomy).

1.2 Problem Statement

The goal of our work is to provide a JAVA API for an agent based transaction manager in MDBMS which has the following behavior:

1. A centralized transaction manager runs a client process, controls the execution of concurrent global transactions.
2. Transaction manager submits subtransactions of the global transaction to AGENT programs which are server processes that run at each site.
3. Transaction manager and AGENTS communicate by using JAVA sockets; can exchange messages to accommodate the protocol described in [10].
4. Transaction manager preserves global atomicity and coordinates the global commit and recovery actions.
5. System functions as desired even in the presence of failures.
6. Timeout mechanism is used to deal with deadlocks.
7. Finally test case scenarios can be used to illustrate the system functionality.

Such an API provides an environment where existing protocol can be verified for correctness and the system is also adaptable to modifications which are useful in refining the protocol.

An overview of related concepts in presented in the following chapters. Chapter 2 describes fundamentals of transaction management; Chapter 3 describes several issues related to transaction management. Chapter 4 describes JAVA Socket programming concepts used to establish communication between TM and AGENT. Chapter 5 describes the overall architecture deployed; Chapter 6 describes the design issues and implementation details. Chapter 7 explains the working of the protocol in 3 different test case scenarios. Chapter 8 briefly discusses related work and Chapter 9 concludes with future enhancements.

2. TRANSACTION MANAGEMENT

A transaction processing system (TPS) includes one or more databases, software for managing the transactions, and the transactions. A transaction consists of one or more requests, and (or) updates to a system. A simple transaction is in the form:

1. Begin the transaction
2. Execute requests or updates
3. Commit or Roll-back

A transaction is a very special kind of program that executes within an application in which a database models the state of some real-world enterprise [12]. For example, if the enterprise is a bank, then the value of the balance attribute of your account is the net amount of money held for you in that account by the bank. The job of the transaction is to maintain this model as the state of the enterprise changes. Thus, whenever the state of the real world changes, a transaction is executed that updates the database to reflect the change. A transaction can perform one or more of the following

1. Update a database to reflect the occurrence of an event

A transaction can update a database to reflect the occurrence of a real-world event that affects the state of the enterprise the database is modeling. An example is a deposit transaction at a bank; here the event is that customer gives teller some cash to deposit. Once the event occurs, the transaction updates the customer's account in the database to reflect the deposit.

2. Ensure that one or more events can occur

A transaction can ensure that one or more real-world events can occur. An example is a withdrawal transaction at an automated teller machine (ATM). The transaction actuates the mechanical device that dispenses the cash, and the debit event occurs if and only if the transaction is successfully completed.

3. Return information

A transaction can return information derived from the database about the current state of the enterprise. An example is a transaction that displays a customer's account balance. A single transaction can perform all the 3 functions. For example, a deposit transaction might

1. Update the database in response to an event where the customer gives cash to the teller.
2. Initiate an event in which a deposit slip is printed if and only if the transaction completes successfully.
3. Display current account balance of the customer from the database.

The execution of transactions is constrained by certain properties because of the requirement that a transaction processing application must maintain an accurate model of the state of the enterprise. These special properties are referred using the acronym ACID: Atomic, Consistent, Isolated, and Durable, described in detail in subsection 2.2

2.1 Classification of Transactions

To deal with complex transactions, many transaction processing systems provide mechanisms that impose structure on transactions or break up a single task into several related transactions. The following are some structuring mechanisms from an application designer point of view.

1. Flat Transactions

A flat transaction is one which has the form

```
Begin_Transaction();
        Statements;
Commit();
```

A flat transaction contains a sequence of operations that satisfies the ACID properties. The transaction is said to be completed when the transaction reflects the changes to the database that were requested.

Limitations of Flat Transactions

A Flat transaction is a simple and clean model for dealing with a series of operations that satisfies ACID properties. However, incase of a failure, a series of successful operations must be undone which is sometimes unnecessary. Let's consider a travel-planning transaction that must make flight reservations for a trip from Atlanta to London. The strategy might be to make a reservation from Atlanta to New York and then reservation from New York to Dallas and finally a reservation from Dallas to London.

Suppose after the first two reservations are made, it is found that there are no seats available on the flight from Dallas to London. There are several approaches to deal with such a scenario, the transaction might abort when it fails to get the reservation from Dallas to London and a subsequent transaction might use a different route through Chicago. But the problem with this approach is that the computation to get the first reservation from Atlanta to New York will be lost and further, a subsequent transaction may find that there are no longer seats available on that flight. Hence a model is needed in which a transaction can preserve partial results. In such a case the transaction can

decide to give up the New York to Dallas reservation or choose a different route say New York to Chicago and Chicago to London where the first reservation from Atlanta to New York is preserved.

Solution with Savepoints

To address the all-or-nothing choice presented by flat transactions, transaction can be decomposed into parts that relate to each other in various ways. Database systems provide savepoints, which are points in a transaction that serve as points of partial rollbacks of the database. A transaction with several savepoints has the following form

```
Begin_Transaction();
        Statement1;
        Save point1 := Create_SavePoint();
        Statement2;
        Save point2 := Create_SavePoint();
        …..     …..
        if (condition) {
                Rollback (Save_Pointi)
        }
Commit();
```

2. Nested Transactions

Nested transactions contain a hierarchy of transactions that allow concurrent processing of subtransactions and have the ability to recover from a subtransaction failure. Here a transaction and all its subtransactions can be viewed as a tree.

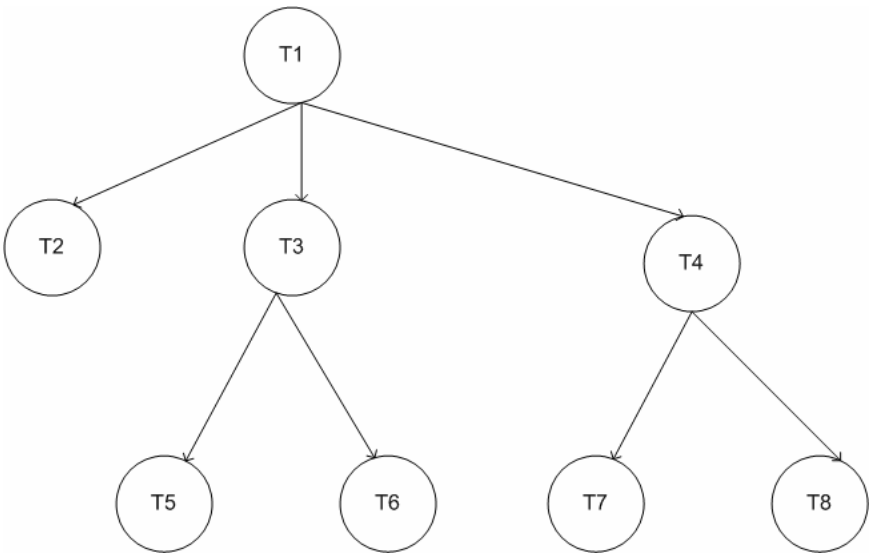


Figure 1: Structure of a Nested Transaction

Let's again consider the travel-planning transaction that must make flight reservations for a trip from Atlanta to London. Say transaction T1 makes reservation for a trip from Atlanta to London. It might create a subtransaction T2 to make a reservation from Atlanta to New York. When T2 completes, it creates a second subtransaction T3 to make a reservation from New York to London. T3 in turn can create might create 2 subtransactions T5, to make a reservation from New York to Dallas and T6, to make a reservation from Dallas to London. T5 and T6 might be specified to execute concurrently. If T6 cannot make the reservation from Dallas to London, it can abort. When T3 finds about T6, it can also abort and causes T5 which is the child of T3 to abort and release the reservation from New York to Dallas. When T1 learns about T3, it can create a new subtransaction T4, to make a reservation from New York to Chicago. T4 in turn might create 2 subtransactions T7, to make a reservation from New York to Chicago and T8, to make a reservation from Chicago to London. T7 and T8 might be specified to execute concurrently. If T4 can commit, T1 can commit and its effect on the database

will be the sum of the effects of T2, T4, T7, and T8. The transaction as a whole is viewed as an isolated atomic unit by other nested transactions.

3. Distributed Transactions

A distributed transaction is a transaction (flat) that is executed on a distributed data; this is often implemented as a two-level nested transaction with one subtransaction per node.

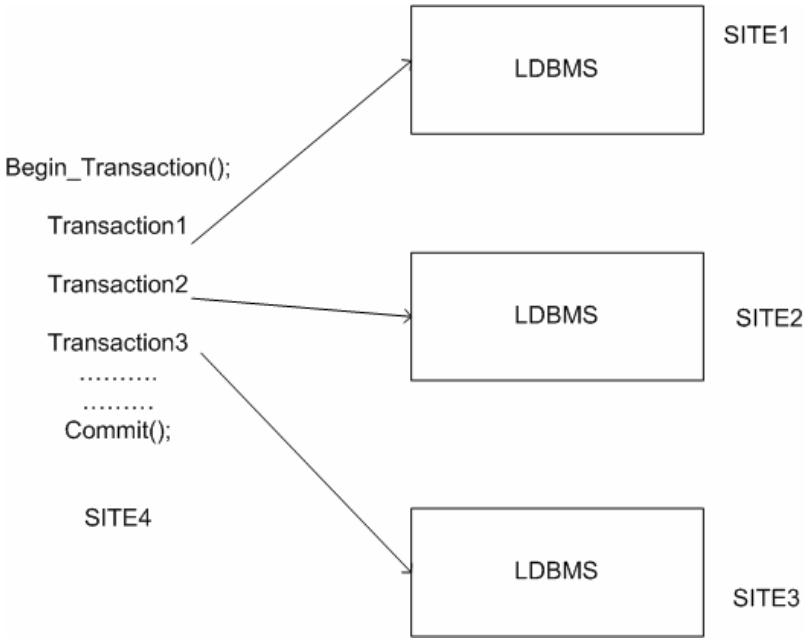


Figure 2: Distributed Transaction

2.2 ACID Properties of Transactions

A transaction processing system must provide certain guarantees concerning how transactions are executed. Transactions must satisfy 4 basic properties called atomicity, consistency, integrity and durability. A transaction either processes all of its changes or none of them, after the execution the system must always be in a valid state.

1. Atomicity:

Refers to the ability of DBMS to ensure that either transaction executes completely or, if it does not execute completely, it has no effect at all.

2. Consistency:

Refers to the database being in a legal state when the transaction begins and when the transaction ends. System either gets a valid new state or remains in the previous state. Execution of a transaction in isolation preserves consistency.

3. Isolation:

Refers to the ability that makes the operations of the transaction isolated from all other operations. Each transaction assumes that it is executed alone in the system and the local DBMS guarantees that intermediate transaction results are hidden from other concurrently executed transactions.

4. Durability:

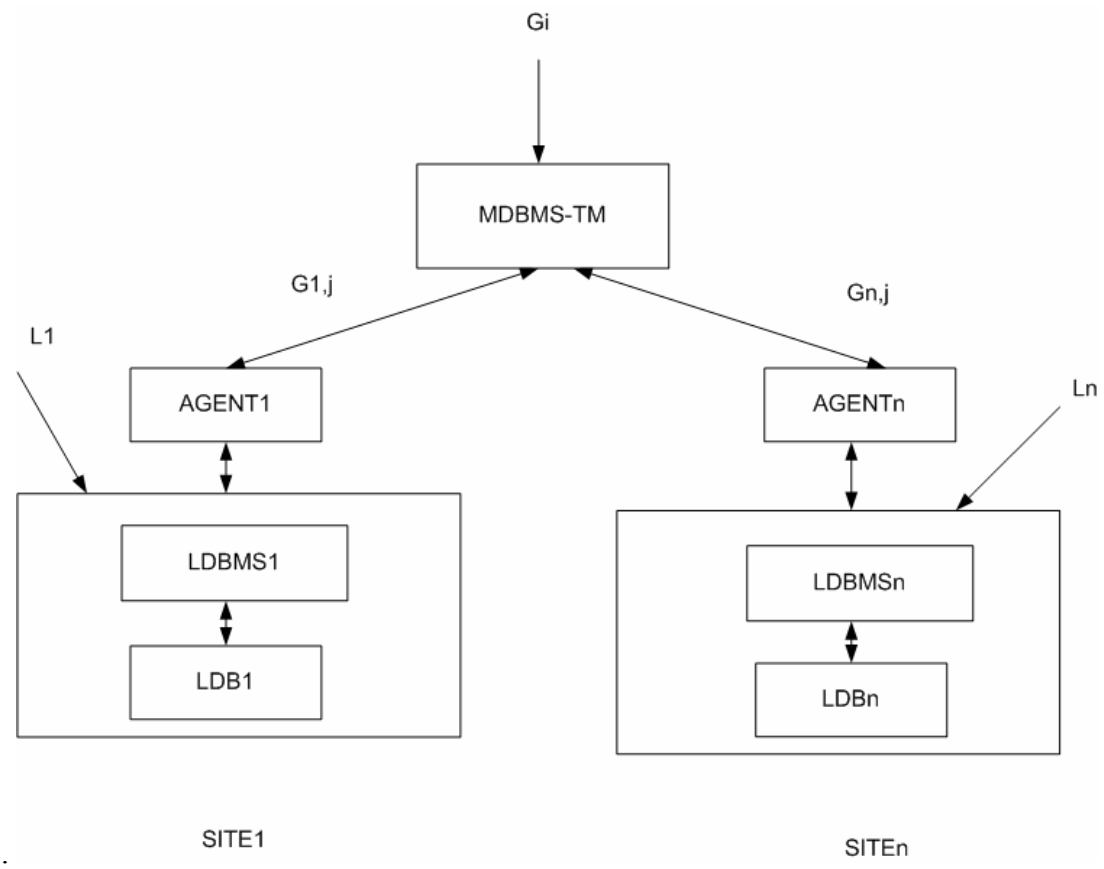
System is left in a valid state even after a system failure. The values changed by the transaction must remain after the transaction is successfully completed.

3. ARCHITECTURE

In this chapter we present the multidatabase system architecture and the responsibilities of multidatabase transaction manager and AGENTS in MDBMS. Then a description of the different state transitions of transaction manager and an AGENT is described using a state-transition-diagram.

3.1 A Multidatabase System Architecture

A multidatabase consists of a set of autonomous existing local databases *LDB1,LDB2,….LDB*$_n$ located at sites *S1,S2,…..S*$_n$ respectively**.**



.

Figure 3: A Multidatabase System Architecture

L_i denotes a local transaction issued and executed locally at site S_i, G_i denotes a global transaction and $G_{i,j}$ denotes a subtransaction of G_i submitted to $LDBMS_j$.

3.2 Multidatabase Transaction Manager (MDBMS-TM or TM) & AGENTS

TM submits subtransactions of a global transaction to the appropriate LDBMS via AGENTS. TM controls the execution of concurrent global transactions to preserve the correct order of execution. Also TM guarantees global atomicity, coordinates global commit and recovery actions.

An AGENT is a component of MDBMS that runs at each site. It can be thought of as an application from a local DBMS point of view. AGENTS receive subtransactions from TM, submit them to LDBMS and send the results to TM.

3.3 State Transitions of TM & AGENTS

In the figures given below ovals in the figure denote the states and the state transitions are represented by edges. The terminal states are represented by concentric ovals. The numerator on the labels of the edges represent the event for the state transition which is a received message and the denominator represents the result of the state transition which is the message sent.

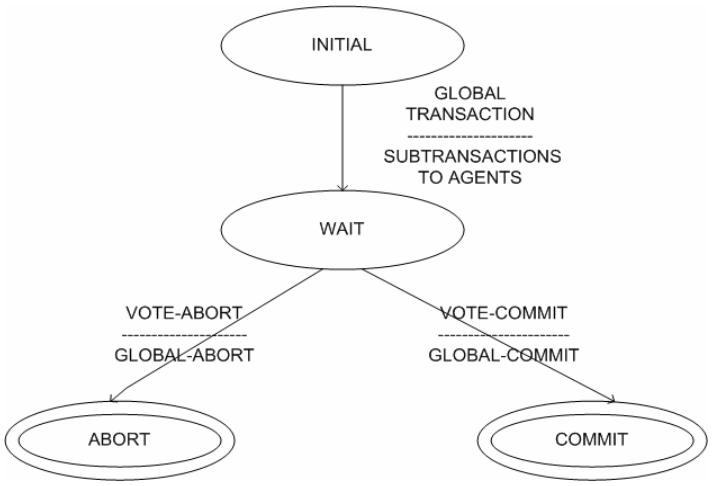


Figure 4: State Transitions of TM

State Transitions of TM

When the TM is in the INITIAL state, it receives the global transaction, sends the subtransactions to respective AGENTS and enters the WAIT state. TM remains in WAIT state until it receives a VOTE-COMMIT message or a VOTE-ABORT message from all the AGENTS. If no message is received from an AGENT then TM assumes that to be VOTE-ABORT. TM then moves from WAIT state to COMMIT state if all the messages received from AGENTS are VOTE-COMMIT else moves to ABORT state.

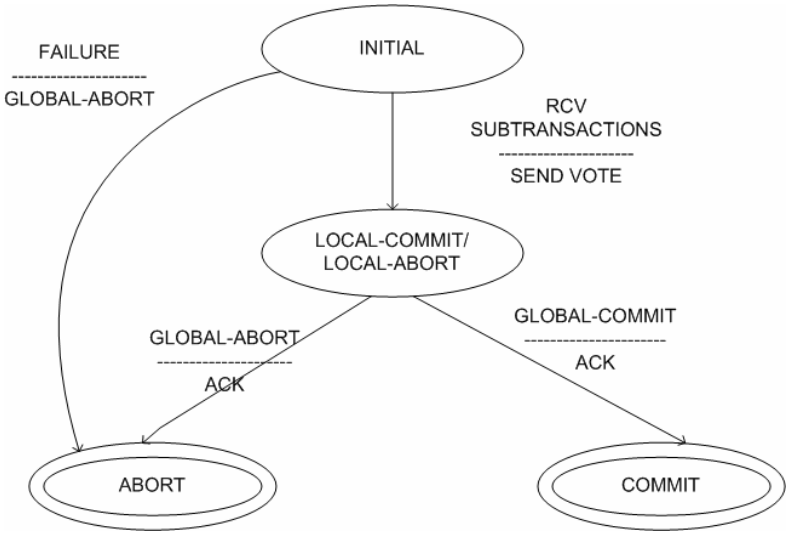


Figure 5: State Transitions of an AGENT

State Transitions of an AGENT

All the AGENTS are initially in the INITIAL state. AGENTS after receiving the subtransactions execute the subtransactions and send a vote which is either a VOTE-COMMIT or VOTE-ABORT and enter the LOCAL-COMMIT/LOCAL-ABORT state. During the INITIAL state, in the event of any failure AGENT enters ABORT state. AGENTS remain in LOCAL-COMMIT/LOCAL-ABORT state until it receives a GLOBAL-COMMIT message or a GLOBAL-ABORT message from TM. If no message is received from TM then AGENT assumes that to be GLOBAL-ABORT message.

AGENT then sends an ACK message to TM saying that the message is received and moves from LOCAL-COMMIT/LOCAL-ABORT state to COMMIT state if the message received from TM is GLOBAL-COMMIT else moves to ABORT state.

4 MULTIDATABASE TRANSACTION MANAGEMENT ISSUES

The TM should guarantee the ACID properties of global transactions should be able to recover from any type of system failures and must ensure deadlock-free execution of global transactions. In this section we discus about concurrency control mechanisms that must be adopted by the TM in order to facilitate multiple transaction execution at the same time. When different LDBMSs use different concurrency control protocols, it raises global serializability issue. Issues related to global serializability are described which is then followed by global recovery and deadlock problems.

4.1 Concurrency Control

Transactions contain a set of READ and WRITE operations. Two operations are said to be conflicting if

1. They belong to two different transactions,
2. They access the same data,
3. And at least one of them is a WRITE operation.

A concurrency control algorithm controls the execution order of conflicting operations such that serializability property is maintained. The goal of concurrency control is to ensure that transactions behave as if they are executed in isolation. Conflict serializability is the conventional concurrency control correction criteria; it is adopted as the global concurrency control correctness criteria. Each local database system has its own concurrency controller or scheduler, hence to integrate several local database systems in

to a multidatabase system; there is a need for a global concurrency controller which gives rise to a hierarchical structure of global concurrency control.

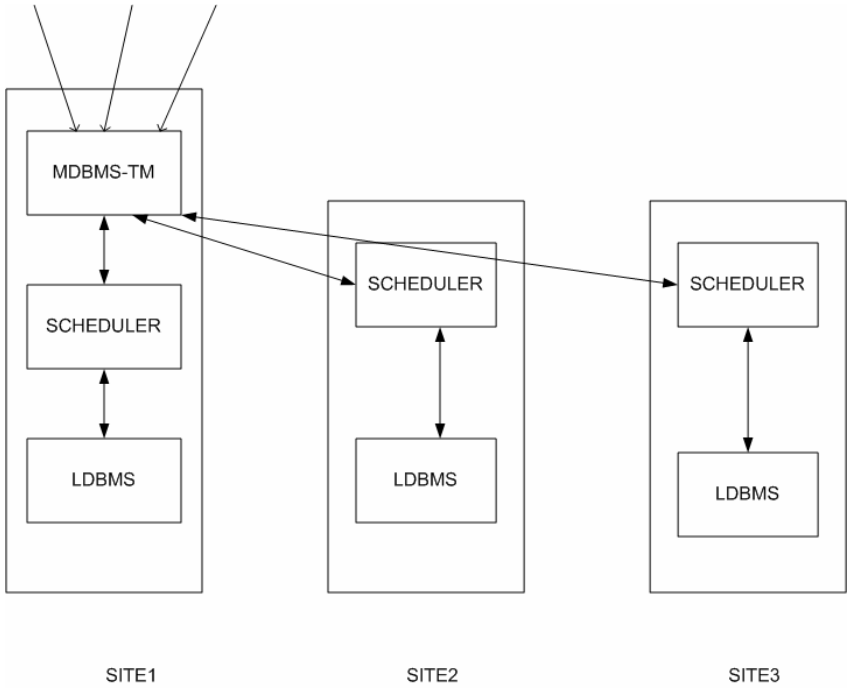


Figure 6: Concurrency control by TM for Distributed Transactions

At the lower level, schedulers maintain local serializability at local sites, at the higher level MDBMS-TM maintains global serializability. MDBMS-TM must determine the serialization order of global transactions at each site without violating local autonomy. The serialization order must ensure consistency with all the schedulers at the local databases. Different concurrency control protocols like 2 Phase Locking (2PL), Time Stamp Ordering, Serialization Graph Testing, and Optimistic control etc; can be used.

4.2 Global Serializability in Multidatabases

A MDBS is a collection of local databases located at sites s_1, s_2, s_3…….s_m, each of which may follow a different concurrency control protocol. This heterogeneity makes it difficult

to ensure global serializability in MDBS environment. Each of the LDBMS is a pre-existing database system which cannot be modified and so

1. LDBMS may not communicate any information related to concurrency control to TM.
2. TM is unaware of the indirect conflicts between global transactions because local transactions are executed at the LDBMS.

Consider a collection E of transactions T_1, T_2,……..,T_n, the goal is to have serializable execution of E, where transactions in E are executed concurrently according to some schedule S. A global schedule S is the set of all operations belonging to local and global transactions with a partial order $<_s$ on them. The local schedule at a site s_k denoted by S_k is the set of all operations belonging to local and global transactions that execute at s_k with a total order $<_{sk}$ on them.

Two operations OP1(Ti, X) and OP2(Tj, X) on the same data item X may conflict incase of

1. *READ-WRITE conflict (RW):* One operation is a READ while other is a WRITE operation on the same data item.
2. *WRITE-WRITE conflict (WW):* Both the operations are WRITE on the same data item.

READ-WRITE and WRITE-WRITE conflicts can be synchronized independently as long as the total ordering of transactions is consistent with both types of conflicts.

Serialization Functions

Serialization functions as mentioned in [13] can be used to ensure global serializability in a MDBS environment. Let Γ_k be the set of all global transactions in S_k, a serialization

function for s_k, serfn, is a function that maps every transaction in Γ_k to one of its operations such that for any pair of transactions T_i, $T_j \in \Gamma_k$, if T_i is serialized before T_j in S_k then $serfn(T_i) <_{sk} serfn(T_j)$.

For example if Time Stamp Ordering protocol is used at site s_k and the LDBMS at the site s_k assigns timestamps to transactions when they begin execution, then the function that maps every transaction $T_i \in \Gamma_k$ to T_i 's begin operation is a serialization function for s_k. There may be multiple serialization functions for a site, for example if 2PL protocol is used at site s_k then a possible serialization function for s_k maps every transaction $T_i \in \Gamma_k$ to the operation tat results in T_i obtaining its last lock. Serialization functions do not exist for sites following certain protocols like Serialization Graph Testing, in such cases serialization functions can be introduced by external means by forcing conflicts between transactions [14].

Global Serializability using serialization functions

Global Serializability [14] is assured if there exists a total order on the global transactions such that at each site s_k, for all pairs of global transactions G_i, G_j executing at site s_k if $serfn(G_i) <_{sk} serfn(G_j)$ then G_i is before G_j in total order.

4.3 Recovery from Failures

Failures of a system may result in loss of information; hence MDBMS should provide a way to recover from an inconsistent database state automatically without any human intervention. For global transaction aborts, recovering multidatabase consistency means undoing the effects of locally committed subtransactions that belong to the aborted global transaction. Also the effects of transactions which have accessed objects updated by aborted global transactions must be preserved. Consistency here means to restore the

most recent global transaction consistent state. Recovery from failures may do one or more of the following [10]

1. Redo the effects of aborted subtransaction
2. Retry (or Resubmit) the aborted subtransaction
3. Compensate (Undo the effects of) the committed subtransaction

4.4 Global Deadlock Problem

Let's consider a MDBS where each LDBMS uses a locking mechanism to ensure local serializability. Here we assume that each LDBMS has a mechanism to detect and recover from deadlocks. Due to the design autonomy LDBMS may not wish to exchange their control information and therefore TM will be unaware of the global deadlock. Also MDBMS is unaware of local transactions and therefore will be unaware of the deadlock. In such cases there is a possibility of a global deadlock that cannot b detected by TM.

Strategies for dealing with deadlocks

Timeout strategy can be used to deal with deadlocks where after a specified amount of time an attempt will be made for successful execution of transactions. Deadlock detection may also be an option, where TM can devise a strategy for approximating the union of the local wait-for-graphs at different sites. If a deadlock exists then there will be a cycle in the approximate wait-for-graph but the converse is not true. A cycle in the approximate wait-for-graph that is not a real deadlock is called a false deadlock. A deadlock detection scheme such as delaying the addition of the arc $T_i \rightarrow T_j$ where T_i could be waiting for T_j for some threshold amount of time could reduce the likelihood of false deadlocks.

4.5 Issues Related to Global Atomicity

A multidatabase commit and recovery protocol must address the following situations as a consequence of autonomy of the LDBMSs as mentioned in [10]:

Unilateral abort of subtransactions due to site or LDBMS failure:

LDBMSs cannot distinguish subtransactions from local transactions. A LDBMS may restart because of a site failure, in such a case local recovery procedure of LDBMS rolls back all uncommitted subtransactions as well as uncommitted local transactions. This may take place even if the global transactions that issued those subtransactions have already decided to commit globally.

Unilateral abort of subtransactions due to commit operation failures:

A subtransaction in a LDBMS may fail at commit operation, this results in a globally inconsistent state.

Intermediate results:

The recovery action of MDBMS is also a transaction that has no connection with the failed subtransactions, from a LDBMS point of view. Other transactions may view intermediate results (with respect to a global transaction) after a unilateral abort of a subtransaction occurs, but before initiation of some recovery action and completion. This may result in inconsistency.

5. SOCKET PROGRAMMING

To accommodate the protocol described in [10] TM and AGENTS must communicate with each other. A JAVA socket is one possible way to establish communication between TM and AGENTS. JAVA sockets are flexible, cause low network traffic and can be easily implemented for general communication. In this section we provide a basic knowledge of Sockets in JAVA [15] and explain how TM and AGENTS can communicate via Sockets.

5.1 Sockets

A socket is a software endpoint that establishes bi-directional communication between a server program and one or more client programs. A socket associates the server program with a specific hardware port on the machine where it runs, so that any client program running in the network with a socket associated with the same port can communicate with the server program.

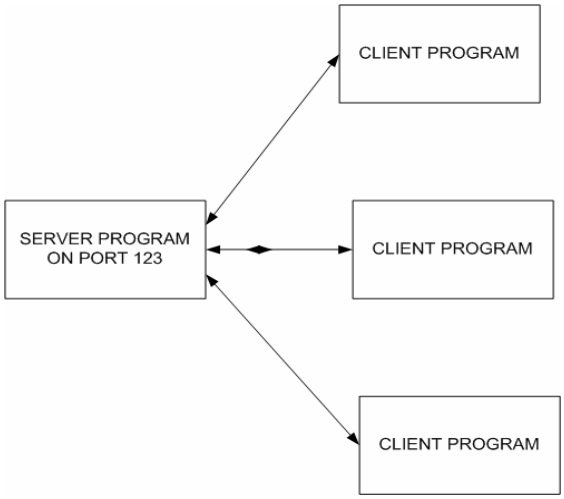


Figure 7: Client –Server communication using Sockets

Requests from multiple clients can be handled by making the server program multithreaded. A multithreaded server creates a separate thread for each accepted communication from client. A thread is a sequence of instructions that can run independent of the program and other threads, thus multithreaded server can continue listening for requests from other clients.

Communication Protocols

There are two communication protocols that can be used for socket programming called Datagram communication and Stream communication.

1. Datagram Communication

The Datagram communication protocol known as User Datagram Protocol (UDP) is a connectionless protocol which means that every time datagrams are sent, local socket descriptor and receivers socket address must also be sent. Here additional data must be sent each time a communication is made. There is a size limit of 64 Kilobytes on the datagrams that can be sent. This is an unreliable protocol meaning, there is no guarantee that the datagrams sent will be received in the same order by the receiver. All the available data can be read immediately in the order in which they are received. UDP is often used to implement client/server applications in distributed systems.

2. Stream Communication

The Stream communication protocol known as Transfer Control protocol (TCP) is a connection-oriented protocol. To communicate over TCP a connection must be established between the pair of sockets. Socket that listens for a connection request is usually called a server and the socket that makes a request is called

client. Once the connection is established data can be transmitted in both the directions. There is no limit on the data that can be sent. This is a reliable protocol; it ensures that the packets sent will be received in the order in which they are sent. TCP is used in applications such as remote login and file transfer where a large amount of data has to be sent.

The choice of UDP or TCP depends on the client/server application.

Implementing a Server over TCP

AGENTS uses a server socket object to accept connections from transaction manager and listen on the specified port number.

```
/*To open a server socket*/
ServerSocket serverSock;
serverSock = new ServerSocket (Port Number);
/*To create an input stream*/
input = new DataInputStream (serverSock.getInputStream());
/*To create an output stream*/
output = new DataOutputStream (serverSock.getOutptStream());
```

Implementing a Client over TCP

Transaction manager uses a socket object on a particular port to connect to the AGENTS which are listening on the same port.

```
/*To open a client socket*/
Socket clientSock;
clientSock =   new Socket ("Machine Name", Port Number);
/*To create an input stream*/
input = new DataInputStream (clientSock.getInputStream());
/*To create an output stream*/
```

```
Output  = new DataOutputStream
(clientSock.getOutputStream());
/* To close sockets*/
clientSock.close();
serverSock.close();
```

Transaction manager and AGENTS can exchange messages as long as the connections are not closed.

5.2 Communication between TM and AGENTS using Sockets

With TM located centrally, it controls the execution of global transactions. It communicates with various LDBMS by means of server processes that execute at each site on top of LDBMS. TM runs a client process and communicates with AGENTS which runs server processes at different sites, and can exchange messages needed for achieving global atomicity. The basis for this type of communication is a JAVA Socket class on the client side where TM creates a Socket object for each port to establish connection to the server on the specified port. Then it creates an InputStream to read global transaction and a PrintStream to write lines of text to socket's OutputStream. On the server side each AGENT uses the ServerSocket class to accept connections from TM. When TM connects to the port that a ServerSocket is listening on, the ServerSocket allocates a new Socket object which is connected to the port on which the TM is communicating. The server can now go back and listen to additional client connections on the ServerSocket. TM and AGENTS can exchange messages as long as the connection is open.

6. DESIGN AND IMPLEMENTATION

In this section we discuss the responsibilities of TM and AGENTS, since we focus on the commit and recovery aspect of MDBS, next we discuss about the commit, termination and recovery protocols followed by the TM and AGENTS.(needs to add**0

6.1 Functions of Multidatabase TM and AGENT

Multidatabase Transaction Manager (TM)

The TM parses the global transaction received, decides sites to which connection is to be established and opens connections to respective sites. It then sends the corresponding subtransactions to the AGENTS at which subtransactions have to be executed. TM controls the order of execution of subtransactions by a unique *transaction_ID* assigned to each subtransaction. TM waits for a *vote* which is either a *vote_abort* or *vote_commit* from each AGENT, finally sends a *g_commit* or *g_abort* message to each AGENT and receives an *acknowledgement* message from each AGENT. TM coordinates the global commit and recovery actions.

AGENT

An AGENT is a component of MDBMS that runs at each site. AGENTS receive subtransactions from TM, submit them to LDBMS. Each AGENT then sends a *vote* back to the TM and waits for *g_commit* or *g_abort* message from TM to make the final decision to either *commit* or *rollback* the subtransaction to preserve global atomicity. They finally send an *acknowledgement* message to TM.

6.2 Commit Protocol

TM Commit Protocol

TM sends subtransactions to all the AGENTS that participate in the execution of the global transaction, and waits for a final decision.

1. If every *vote* from the AGENTS is *vote_commit*, sends a *g_commit.*
2. Even if one *vote* from the AGENTS is *vote_abort*, sends a *g_abort.*
3. Each AGENT sends an *acknowledgement* to TM and that is the end of the transaction.

Agents Commit Protocol

1. AGENTS receive subtransactions; lock the tables on which transactions are being performed to preserve consistency.
2. Sends either a *vote_commit* if subtransaction can be executed in its entirety or a *vote_abort* if the subtransaction could not be executed completely.
3. Receives a *g_commit* or *g_abort* message, releases locks, and then sends an *acknowledgement* message to TM.

6.3 Termination Protocol

Timeout mechanism is used for termination of AGENTS. AGENTS can timeout in two states: INITIAL and Local COMMIT/ABORT.

Timeout in INITIAL State:

In this state AGENTS are waiting for subtransactions from TM. AGENT can unilaterally abort the subtransaction due to a timeout. If the subtransaction arrives after the AGENT is aborted, no vote is sent from the AGENT and TM treats it as a *vote-abort* and TM proceeds further.

Timeout in Local COMMIT/ABORT State

In this state AGENT has already sent a *vote_commit* or *vote_abort* and is waiting for a *g_commit* or *g_abort* from TM. If a timeout occurs, AGENTS don't receive *g_commit* or *g_abort* from TM, and the subtransaction is not committed.

6.4 Recovery Protocol

TM fails in INITIAL State

When TM fails in this state, AGENTS don't receive subtransactions and so nothing happens at the AGENTS side. Thus system remains in a valid state.

TM fails in WAIT state

When TM fails in this state, waiting for *votes* from AGENTS, they don't receive any message from TM to either commit or rollback. Locally committed subtransactions are rolled back when no message is received from TM. Thus system remains in a valid state.

TM fails in COMMIT or ABORT state

In this state, TM receives *votes* from AGENTS but fails to send either *g_commit* or *g_abort*, AGENTS don't receive any message from TM to either commit or rollback. Locally committed subtransactions are rolled back when no message is received from TM. Thus upon recovery, the system remains in a valid state.

Agent Site fails in INITIAL state

When an agent site fails or LDBMS at that site fails, it receives no subtransactions from TM and so upon recovery, the system remains in a valid state.

AGENT site fails in Local COMMIT/ABORT state

When an agent fails in this state, it receives no message from TM. Then the subtransaction is not executed and TM doesn't receive any acknowledgement and upon recovery the system is in a valid state.

AGENT site fails in COMMIT or ABORT state

These states represent termination states; hence no special actions are required for recovery. The system remains in a valid state.

7. TEST CASE SCENARIOS

We now discuss 3 different test case scenarios where the global transaction commits, rollbacks and how site failures are handled. Let's say there is a global transaction G1 containing subtransactions that are to be executed at sites SITE1, SITE2, SITE3 .The LDBMS at SITE1 is DB1 and at SITE2 is DB2 and at SITE3 is DB3 respectively. Each global transaction submitted to the TM is of the form:

SITENAME: TRANSACTION_ID\\QUERY

Here an UPDATE query is of the form:

UPDATE DBNAME.Tablename SET <expression> WHERE <condition>

A SELECT Query is of the form:

SELECT <expression> FROM DBNAME.Tablename [,DBNAME.Tablename….] WHERE <condition>

7.1 Commit

Let's now consider a case where the Global transaction commits. Say Gtransaction.txt contains the following:

```
SITE1:T1\\UPDATE DB1.PARTS SET price = 1010 WHERE pid = 9
SITE1:T2\\SELECT pid, price FROM DB1.parts, DB2.products
WHERE pid = 2
SITE2:T1\\UPDATE DB2.products SET qty = 900 WHERE pno = 9
SITE2:T2\\SELECT pname, price FROM DB1.parts
SITE3:T1\\SELECT * FROM DB3.students
```

```
query3: SITE2:T2\\SELECT pname,price FROM DB1.parts
-----
query4: SITE3:T1\\SELECT * FROM DB3.students
-----
********************End of Queriese***********************************
PortNum:6030
SITE:SITE3
SiteHashTable={SITE3_T1={DB3={STUDENTS=select * from STUDENTS}}}
****Sockets Creation***************
Sock 0 created on port 6030
******End of Sockets Creation*************
Vote From AGENT0:commit
***************************
PortNum:6010
SITE:SITE1
SiteHashTable={SITE1_T2={DB2={PRODUCTS=select * from PRODUCTS}, DB1={PARTS=selec
t * from PARTS}}, SITE1_T1={DB1={PARTS=UPDATE PARTS SET PRICE = 1010 WHERE PID =
 9}}}
****Sockets Creation***************
Sock 1 created on port 6010
******End of Sockets Creation*************
Vote From AGENT1:commit
***************************
PortNum:6020
SITE:SITE2
SiteHashTable={SITE2_T2={DB1={PARTS=select * from PARTS}}, SITE2_T1={DB2={PRODUC
TS=UPDATE PRODUCTS SET QTY = 900 WHERE PNO = 9}}}
****Sockets Creation***************
Sock 2 created on port 6020
******End of Sockets Creation*************
Vote From AGENT2:commit
***************************
*********************************************************
Message to AGENTS = commit
Acknowledgement From AGENT0:COMMIT
Acknowledgement From AGENT1:COMMIT
Acknowledgement From AGENT2:COMMIT

******************
[~/Thesis/SampleJava][10:07pm]
```

→{SITE1_T2={DB2={PRODUCTS=select * from PRODUCTS},
DB1={PARTS=select * from PARTS}},
SITE1_T1={DB1={PARTS=UPDATE PARTS SET PRICE = 1010 WHERE PID
= 9}}} to SITE1

```
agg2.class                      testparser.java
agg2.java                       testqparser.class
agg3$1createConnection.class    testqparser.java
[~/Thesis/SampleJava][10:04pm] java AGENT1




*******************************************
AGENT1 starting ...to communicate with TM on port = 6010
**********TM accessed***************
SITE1_T1
-------

Database:DB1Table:PARTS
Query: UPDATE PARTS SET PRICE = 1010 WHERE PID = 9
Locking Table...: PARTS
SITE1_T2
-------

Database:DB2Table:PRODUCTS
Query: select * from PRODUCTS
Database:DB1Table:PARTS
Query: select * from PARTS
Vote from AGENT1 = commit
Message recieved from TM = commit
Sending Acknowledgement of received message to TM









*******************************************
[~/Thesis/SampleJava][10:07pm]
```

→{SITE2_T2={DB1={PARTS=select * from PARTS}},

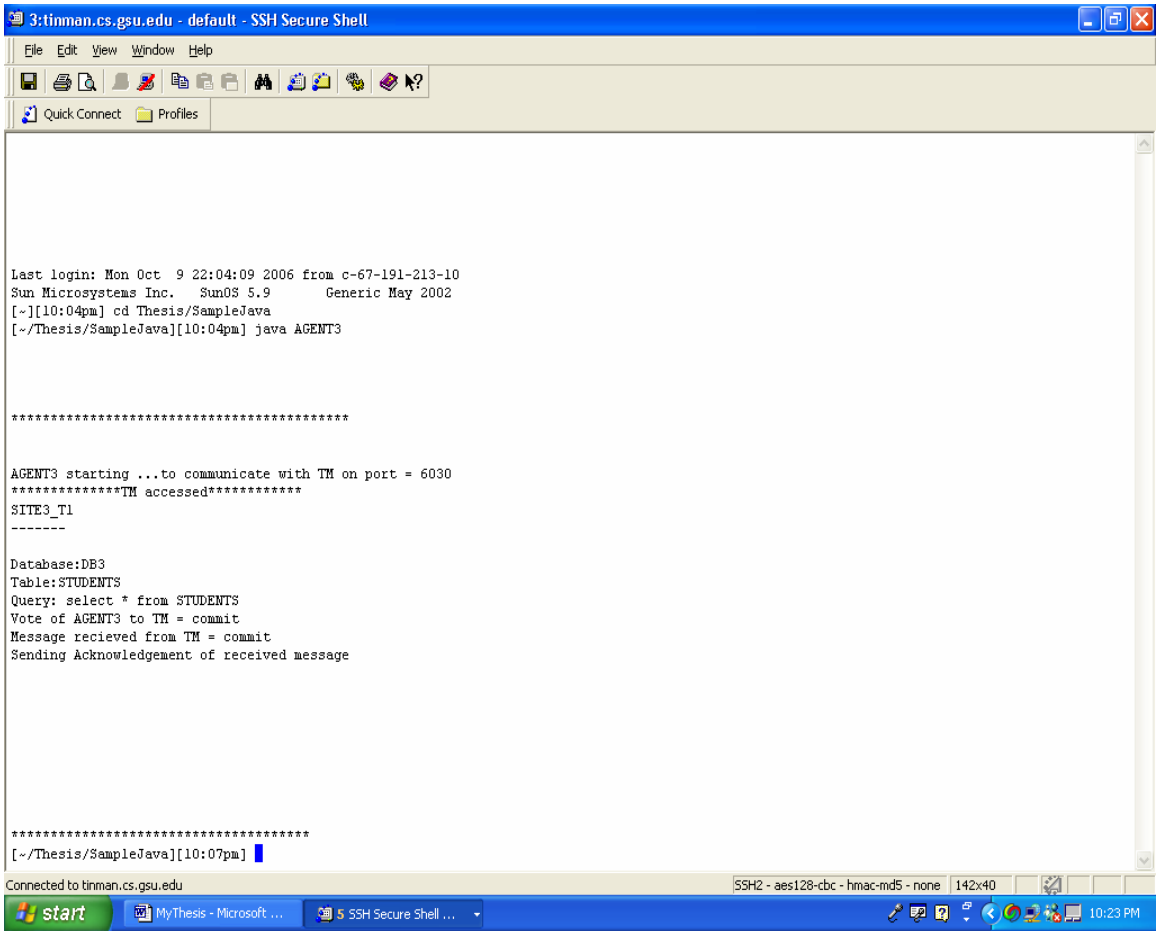
SITE2_T1={DB2={PRODUCTS=UPDATE PRODUCTS SET QTY = 900 WHERE

PNO = 9}}} to SITE2

```
2:tinman.cs.gsu.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect  Profiles

        at java.net.ServerSocket.<init>(ServerSocket.java:185)
        at java.net.ServerSocket.<init>(ServerSocket.java:97)
        at AGENT1.main(AGENT1.java:154)
[~/Thesis/SampleJava][10:06pm] ava AGENT2
ava: Command not found.
[~/Thesis/SampleJava][10:07pm] java AGENT2



*******************************************


AGENT2 starting ...to communicate with TM on port = 6020
*******************TM accessed******************8
SITE2_T1
-------

Database:DB2Table:PRODUCTS
Query: UPDATE PRODUCTS SET QTY = 900 WHERE PNO = 9
Locking Table...: PRODUCTS
SITE2_T2
-------

Database:DB1Table:PARTS
Query: select * from PARTS
Vote of AGENT2 = commit
Message recieved from TM = commit
Sending Acknowledgement of received  message









*************************
[~/Thesis/SampleJava][10:07pm]
Connected to tinman.cs.gsu.edu        SSH2 - aes128-cbc - hmac-md5 - none   142x40
start   MyThesis - Microsoft ...   5 SSH Secure Shell ...   10:21 PM
```

→ {SITE3_T1={DB3={STUDENTS=select * from STUDENTS}}} to SITE3

Agents receive the individual subtransactions and LOCK the tables on which there is an UPDATE, execute the queries. Here VOTE = COMMIT is sent from all the SITES.TM then sends a GCOMMIT message to all the Agents. Then the transaction is committed.



7.2 Abort

Let's now consider a case where the Global transaction rolls back to its previous safe state. Say Gtransaction.txt contains the following:

```
SITE1:T1\\UPDATE DB1.PARTS SET price = 1010 WHERE pid = 9
SITE1:T2\\SELECT pid, price FROM DB1.parts, DB2.products
WHERE pid = 2
SITE2:T1\\UPDATE DB2.products SET qty = 900 WHERE pno = 9
SITE2:T2\\UPDATE DB1.parts SET price = 2020 WHERE pno = 9
SITE2:T3\\SELECT pname, price FROM DB1.parts
SITE3:T1\\SELECT * FROM DB3.students
```

4:tinman.cs.gsu.edu - default - SSH Secure Shell

File Edit View Window Help

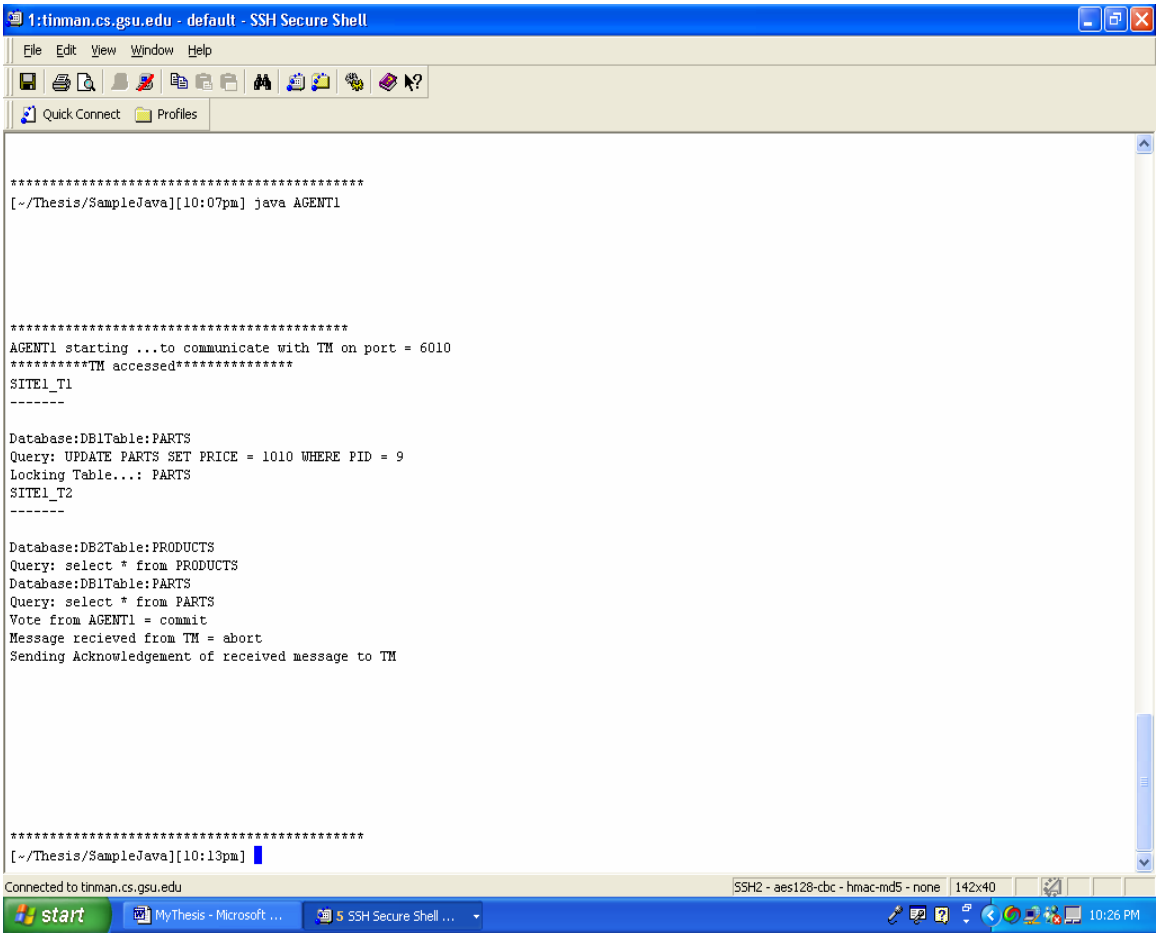
Quick Connect Profiles

```
-----
query4: SITE2:T3\\SELECT pname,price FROM DB1.parts
-----
query5: SITE3:T1\\SELECT * FROM DB3.students
-----
********************End of Queriese************************************
PortNum:6030
SITE:SITE3
SiteHashTable={SITE3_T1={DB3={STUDENTS=select * from STUDENTS}}}
****Sockets Creation***************
Sock 0 created on port 6030
******End of Sockets Creation*************
Vote From AGENT0:commit
***************************
PortNum:6010
SITE:SITE1
SiteHashTable={SITE1_T2={DB2={PRODUCTS=select * from PRODUCTS}, DB1={PARTS=select * from PARTS}}, SITE1_T1={DB1={PARTS=UPDATE PARTS SET PRICE
= 1010 WHERE PID = 9}}}
****Sockets Creation***************
Sock 1 created on port 6010
******End of Sockets Creation*************
Vote From AGENT1:commit
***************************
PortNum:6020
SITE:SITE2
SiteHashTable={SITE2_T3={DB1={PARTS=select * from PARTS}}, SITE2_T2={DB1={PARTS=UPDATE PARTS SET PRICE = 2020 WHERE PNO = 9}}, SITE2_T1={DB2={
PRODUCTS=UPDATE PRODUCTS SET QTY = 900 WHERE PNO = 9}}}
****Sockets Creation***************
Sock 2 created on port 6020
******End of Sockets Creation*************
Vote From AGENT2:abort
***************************
*******************************************************
Message to AGENTS = abort
Acknowledgement From AGENT0:ABORT
Acknowledgement From AGENT1:ABORT
Acknowledgement From AGENT2:ABORT

*******************
[~/Thesis/SampleJava][10:13pm]
```

Connected to tinman.cs.gsu.edu SSH2 - aes128-cbc - hmac-md5 - none 142x40

start MyThesis - Microsoft ... 5 SSH Secure Shell ... 10:26 PM

→{SITE1_T2={DB2={PRODUCTS=select * from PRODUCTS},
DB1={PARTS=select * from PARTS}},
SITE1_T1={DB1={PARTS=UPDATE PARTS SET PRICE = 1010 WHERE PID = 9}}} to SITE1

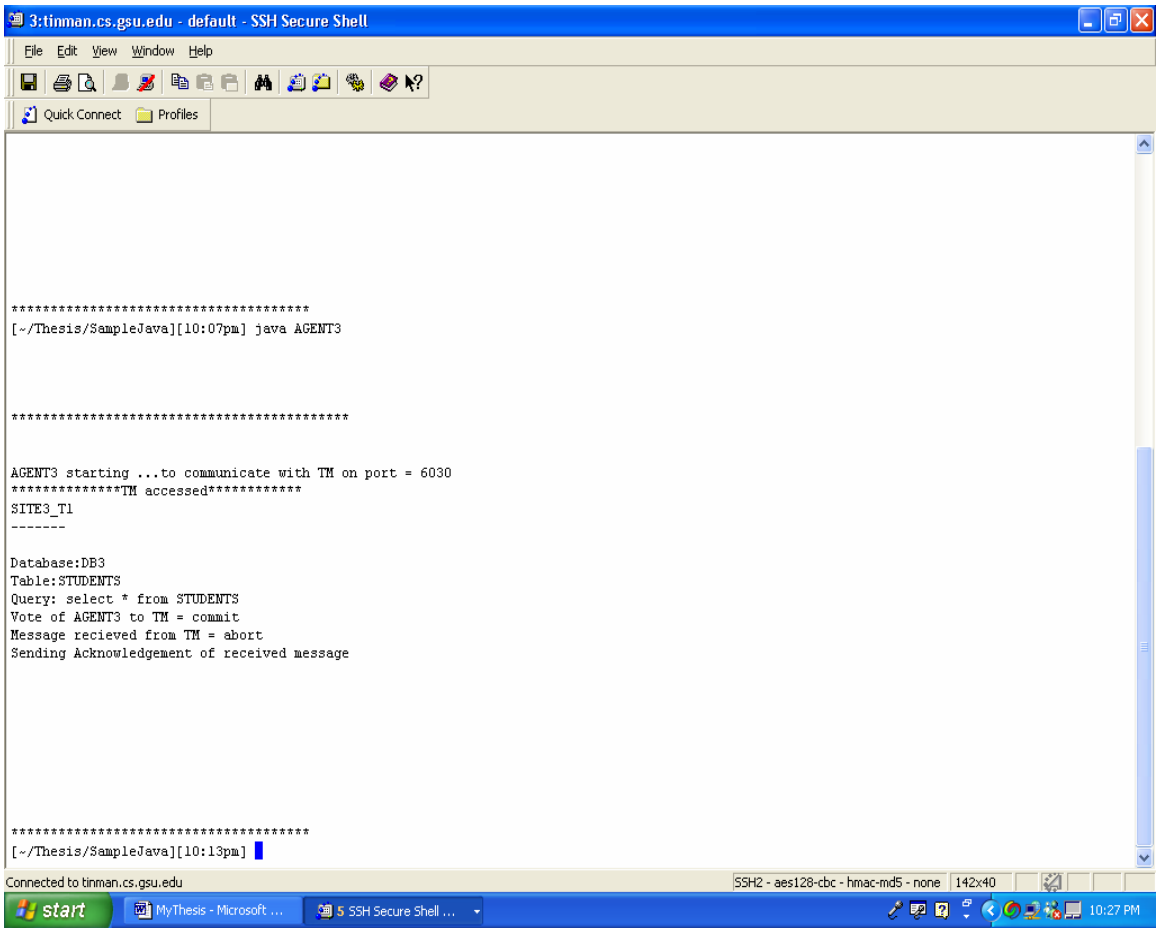


→{SITE2_T3={DB1={PARTS=select * from PARTS}}, SITE2_T2={DB1={PARTS=UPDATE PARTS SET PRICE = 2020 WHERE PNO = 9}}, SITE2_T1={DB2={PRODUCTS=UPDATE PRODUCTS SET QTY = 900 WHERE PNO = 9}}} to SITE2

```
2:tinman.cs.gsu.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect  Profiles

*******************************************


AGENT2 starting ...to communicate with TM on port = 6020
******************TM accessed*****************8
SITE2_T1
-------

Database:DB2Table:PRODUCTS
Query: UPDATE PRODUCTS SET QTY = 900 WHERE PNO = 9
Locking Table...: PRODUCTS
SITE2_T2
-------

Database:DB1Table:PARTS
Query: UPDATE PARTS SET PRICE = 2020 WHERE PNO = 9
Locking Table...: PARTS
Exception in Locking TablesORA-00054: resource busy and acquire with NOWAIT specified

SITE2_T3
-------

Database:DB1Table:PARTS
Query: select * from PARTS
Vote of AGENT2 = abort
Message recieved from TM = abort
Sending Acknowledgement of received  message









*************************
[~/Thesis/SampleJava][10:13pm]
Connected to tinman.cs.gsu.edu        SSH2 - aes128-cbc - hmac-md5 - none   142x40
start   MyThesis - Microsoft ...   5 SSH Secure Shell ...   10:26 PM
```

→ {SITE3_T1={DB3={STUDENTS=select * from STUDENTS}}} to SITE3

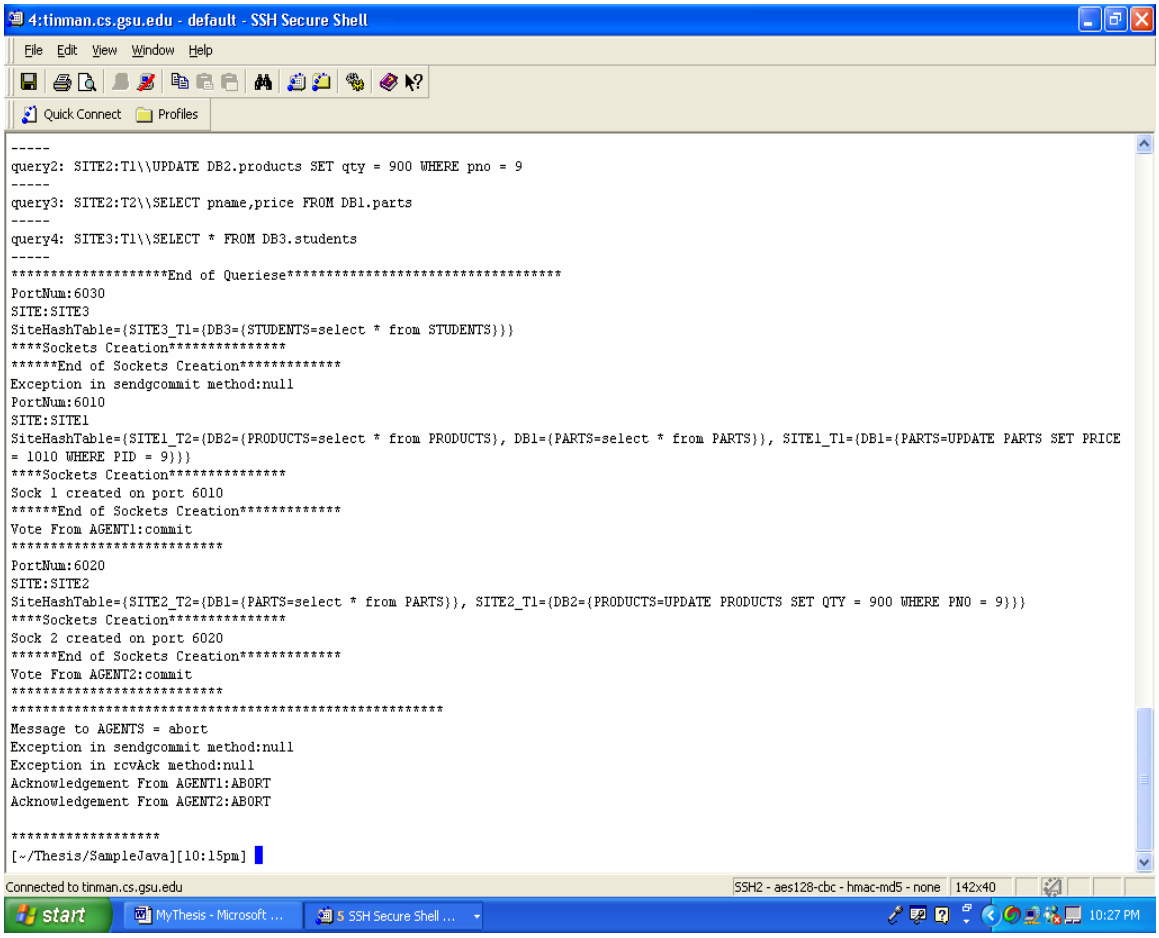
Agents receive the individual subtransactions and LOCK the tables on which there is an UPDATE, execute the queries. Here VOTE = COMMIT is sent from Agents at SITE1 and SITE3, VOTE = ABORT is sent from Agent at SITE2 the SITES.TM then sends a GROLLBACK message to all the Agents. Then the transaction is rolled back to its previous safe state.



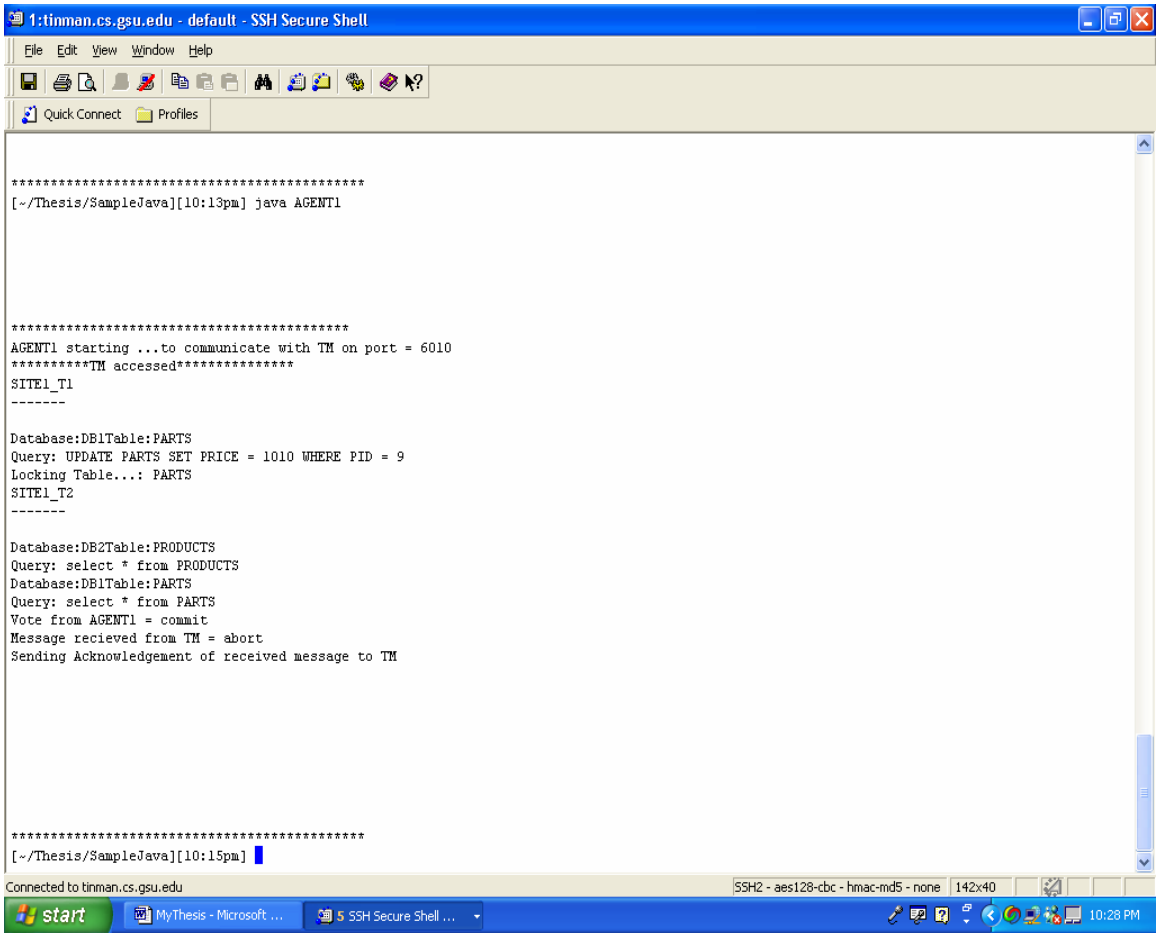
7.3 Site Failure:

Let's now consider a case where the Global transaction rolls back to its previous safe state because of a site failure. Say Gtransaction.txt contains the following:

```
SITE1:T1\\UPDATE DB1.PARTS SET price = 1010 WHERE pid = 9
SITE1:T2\\SELECT pid, price FROM DB1.parts, DB2.products
WHERE pid = 2
SITE2:T1\\UPDATE DB2.products SET qty = 900 WHERE pno = 9
SITE2:T2\\SELECT pname, price FROM DB1.parts
SITE3:T1\\SELECT * FROM DB3.students
```



→{SITE1_T2={DB2={PRODUCTS=select * from PRODUCTS},
DB1={PARTS=select * from PARTS}},
SITE1_T1={DB1={PARTS=UPDATE PARTS SET PRICE = 1010 WHERE PID = 9}}} to SITE1



→{SITE2_T2={DB1={PARTS=select * from PARTS}},

SITE2_T1={DB2={PRODUCTS=UPDATE PRODUCTS SET QTY = 900 WHERE

PNO = 9}}} to SITE2

```
2:tinman.cs.gsu.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect  Profiles

**************************
[~/Thesis/SampleJava][10:13pm] java AGENT2



*******************************************


AGENT2 starting ...to communicate with TM on port = 6020
*******************TM accessed******************8
SITE2_T1
-------

Database:DB2Table:PRODUCTS
Query: UPDATE PRODUCTS SET QTY = 900 WHERE PNO = 9
Locking Table...: PRODUCTS
SITE2_T2
-------

Database:DB1Table:PARTS
Query: select * from PARTS
Vote of AGENT2 = commit
Message recieved from TM = abort
Sending Acknowledgement of received  message









**************************
[~/Thesis/SampleJava][10:15pm]
Connected to tinman.cs.gsu.edu        SSH2 - aes128-cbc - hmac-md5 - none   142x40
start   MyThesis - Microsoft ...   5 SSH Secure Shell ...   10:28 PM
```

→SITE3 failed due to some problems, cannot receive subtransactions from TM.

Here VOTE = COMMIT is sent from SITE1 and SITE2, SITE3 doesn't send any vote. Hence TM assumes VOTE = null. TM then sends a GROLLBACK message to all the Agents. Then the transaction is rolled back to its previous safe state.

8. RELATED WORK

[2] Describes MDBS design problems like concurrency control, commitment, and recovery, since then transaction management in MDBS environment has been studied. Earlier with an assumption that failures don't occur, several approaches to address transaction management issues like concurrency control, commitment and recovery have been proposed [3, 4, 5, 6, and 7]. In a failure free environment serializability of global transactions can be achieved by strict 2PL of the LDBMSs [4, 16]. However that cannot be applied incase of a subtransaction failure which can occur because LDBMSs do not have the ready state that is required to participate in traditional two phase commit protocol [8, 17]. [9] Shows that it is impossible without any violation of local autonomy, to implement global atomic commitment protocol in MDBS environments in the presence of failures. It also shows that strict 2PL used at all LDBMSs as their concurrency control mechanism fails. But to ensure natural integration of several LDBMSs, MDBMS should preserve its local autonomy [1, 11]. Many works achieve atomic commitment incase of failures in multidatabase environment but with certain tradeoffs.

[3, 11, 18] violate local autonomy where users control local transactions so that incomplete results are not accessed until recovery from LDBMS because of a subtransaction failure is completed. [19, 16] have some restrictions on the allowed transactions and data these transactions can access. While some works [16, 18, 20] assume that a subtransaction failure does not occur due to the failure of a commit

operation. [20] Blocks the LDBMSs completely until MDBMS site is recovered from a subtransaction failure or LDBMS failure.

[10] Achieves global atomic commitment and recovery that can handle failures, and also can preserve maximal local autonomy without any restrictions on the multidatabase transactions.

Multidatabase systems based on single multidatabase servers are not realistic as the number of component database systems are increasing. [21] Focuses on the architecture in which the multidatabase system consists of multiple heterogeneous peer servers distributed on a communication network. A global multidatabase request can span multiple servers causing some servers to act as component database systems. The effect of multidatabase composition on global concurrency control algorithms for single server systems has been studied. This approach assumes a homogeneous composite multidatabase system where multiple instances of the same multidatabase server manage disjoint sets of component database systems. Two global concurrency control algorithms called site locking and forced conflicts that ensure multidatabase serializability have been proposed. The site locking algorithm is extended for distributed multidatabase concurrency control which treats the schedulers at each multidatabase server as remote lock managers that receive site lock requests from other multidatabase servers. The forced conflict method is extended for multiple servers by defining a unique totally ordered composite ticket for transactions across multiple multidatabase servers. [22] Shows that forced conflicts or ticket schemes guarantee multidatabase serializability, overcome the problems caused by indirect conflicts. [21] is simplified by assuming that

all multidatabase systems use the same concurrency control algorithm and neither data nor data structures are duplicated or distributed across multidatabase system domains.

Redo, Retry and Compensate mechanisms for ensuring global transaction atomicity have been studied in the literature. The atomicity of a global transaction can be preserved by executing compensating transactions to semantically undo the effects of the globally committed subtransactions (backward recovery approach) [3, 23, 24, 25, 26] or by retrying the aborted globally subtransactions until they commit (forward recovery approach) [19, 27, 28, 29]. In the semantic undo approach, each global transaction of a multisite transaction is associated with a compensating subtransaction. The compensating subtransaction undoes semantically, rather than physically the effect of the subtransaction if the entire multisite transaction aborts. Aborting a global transaction that consumes many resources is too costly if the transaction is long-lived. Hence it is more appropriate to use forward recovery approach by retrying the failed global subtransactions. This method assumes that the execution of a global transaction at one site is independent of its execution at other sites. [30] proposed a recovery strategy that enables MDBMS to deal with failures in multidatabase environments. The recovery strategy, denoted by ReMT consists of a collection of recovery protocols which are distributed among the components of an MDBS. The key advantage of the ReMT strategy is that it can reduce the frequency of global transaction undo after the occurrence of failures, and it is able to deal with several types of failures which may occur in a multidatabase environment.

Deadlock detection problem is closely related to the problem of global database consistency. Ticket methods [7] for ensuring global serializability use a timeout mechanism to solve the deadlock problem [31] shows that the performance of timeout

mechanism is poor when compared to deadlock detection techniques. The deadlock detection algorithm of [32, 33, 34] based on Potential Conflict Graph (PCG) performs better than simple timeout mechanism. PCG describes the potential waiting (direct or indirect conflicts) due to data dependencies. It is proven that an occurrence of a real global deadlock implies an occurrence of a cycle in the PCG but the converse is not true. The cycles in PCG are called potential deadlocks, [31] shows that the only use of the PCG within the extended multidatabase transaction model is insufficient. The transaction model considered here is characterized by the presence of structural dependencies that make the PCG based deadlock detection invalid. A Petri-net denoted by EPC-Net (Extended Potential Conflict Net) models the waiting relations between subtranasctions due to data and/or structural dependencies. Then EPC-net, an effective necessary structural condition for the occurrence of global deadlocks is established, and then a linear time algorithm for potential global deadlock detection in MDBS with an extended transaction model uses EPC-net to detect deadlocks in this approach.

9. CONCLUSION AND FUTUREWORK

In this section we talk about the implementation details and possible design enhancements. We have presented a unique proof-of-concept, a JAVA API for an AGENT Based Transaction Manager that preserves global atomicity. It provides a user friendly interface through which reliable atomic commitment protocol [10] for global transaction execution in multidatabase environment can be visualized.

9.1 Implementation Details

Communication between transaction manager and different agents is achieved using JAVA Sockets [15]. Database *update* operations belonging to a global transaction require locks on the data items before they are submitted to LDBMS. This type of locking can cause a deadlock, to prevent this lock mode is set to *NOWAIT,* this means that if a data item that a transaction wants to lock is already locked by another transaction, the LDBMS rejects the request immediately instead of blocking the transaction. With three different test case scenarios how the protocol works. This helps in further research in this area where atomicity of transactions can be verified for protocol correctness. This has good potential for further improvements and extensions.

9.2 Future Extensions

Future work will include improving the performance of the protocol [10] by reducing its overheads, and extending it will include extending the protocol where TM is distributed. Also recovery protocol [19] can be adapted and compared to the recovery protocol [10] used in the current approach. Currently, the API design is good enough to take a global

transaction and check the protocol correctness. API can further be improved, where it is able to take global transaction independent of the domain structure. This would make the API flexible enough for various domain applications.

10. BIBLIOGRAPHY

[1] Breitbart, Y., Garcia-Molina, H., and Silberschatz, A. 1992. Overview of multidatabase transaction management. *The VLDB Journal* 1, 2 (Oct. 1992), 181-240.

[2] Gligor, V. and Popescu-Zeletin, R. 1986. Transaction management in distributed heterogeneous database management systems. *Inf. Syst.* 11, 4 (Oct. 1986), 287-297

[3] R. Alonso, H. Garcia-Molina, and K. Salem , Concurrency control and recovery for global procedures in federated database systems, *IEEE Data Engineering Bulletin* 5-11 (Sept. 1987).

[4] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, A. Silberschatz, "On rigorous Transaction Scheduling," *IEEE Transactions on Software Engineering* ,vol. 17, no. 9, pp. 954-960, September, 1991

[5] Breitbart, Y. and Silberschatz, A. 1988. Multidatabase update issues. In *Proceedings of the 1988 ACM SIGMOD international Conference on Management of Data* (Chicago, Illinois, United States, June 01 - 03, 1988). H. Boral and P. Larson, Eds. SIGMOD '88. ACM Press, New York, NY, 135-142.

[6] Elmagarmid, A. K. and Du, W. 1990. A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems. In *Proceedings of the Sixth international Conference on Data Engineering* (February 05 - 09, 1990). IEEE Computer Society, Washington, DC, 37-46.

[7] Georgakopoulos, D., Rusinkiewicz, M., and Sheth, A. P. 1991. On Serializability of Multidatabase Transactions Through Forced Local Conflicts. In *Proceedings of the Seventh international Conference on Data Engineering* (April 08 - 12, 1991). IEEE Computer Society, Washington, DC, 314-323

[8] Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987 *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc.

[9] Mullen, J.G.; Elmagarmid, A.K.; Kim, W.: *On the Impossibility of Atomic Commitment in Multidatabase Systems*. In Ng, P., Ramamoorthy, C.,

Seifert,L. and Yeh, R., editors, Proc. of The International Conference on System Integration (ICSI'92), pages 625-634. IEEE Computer Society Press.

[10] Ho-Dong Yoo, Myoung Ho Kim, A Reliable Global Atomic Commitment Protocol for Distributed Multidatabase Systems, *Information Sciences* , p.49-76, 1995.

[11] Soparka, N., Korth, H. F., and Silberschatz, A. 1991. Failure-Resilient Transaction Management in Multidatabase. *Computer* 24, 12 (Dec. 1991), 28-36.

[12] Database Systems: An Application Oriented Approach, Compete Version plus Database Place Student Access Kit (2nd Edition) (Hardcover) by Michael Kifer (Author), Arthur Bernstein (Author), Philip M. Lewis (Author)

[13] Mehrotra, S., Rastogi, R., Breitbart, Y., Korth, H. F., and Silberschatz, A. 1992. The concurrency control problem in multidatabases: characteristics and solutions. In *Proceedings of the 1992 ACM SIGMOD international Conference on Management of Data* (San Diego, California, United States, June 02 - 05, 1992). M. Stonebraker, Ed. SIGMOD '92. ACM Press, New York, NY, 288-297.

[14] Dimitrios Georgakopoulos , Marek Rusinkiewicz , Amit P. Sheth, On Serializability of Multidatabase Transactions Through Forced Local Conflicts, Proceedings of the Seventh International Conference on Data Engineering, p.314-323, April 08-12, 1991

[15] http://java.sun.com/j2se/1.4.2/docs/api/java/net/Socket.html

[16] A. Wolski and J. Veijalainen. "*2pc agent method : Achieving serializability in presence of failures in a heterogeneous multidatabase* ". In Proceedings of PARBASE-90 Conference, 1990.

[17] Ozsu, M. T. and Valduriez, P. 1991 *Principles of Distributed Database Systems*. Prentice-Hall, Inc.

[18] D. Georgakopoulos. "*multidatabase recoverability and recovery* ". In Proc. of the 1st International Workshop on Interoperability in Multidatabase Systems, Kyoto, Japan, 1991.

[19] A. Elmagarmid J. Jing and W. Kim, "Global Committability in Multidatabase Systems," Technical Report CSD TR-91-017, Dept. of Computer Science, Purdue Univ., 1991.

[20] K. BARKER. Transaction Management on Multidatabase Systems. Ph.D. Thesis, Edmonton, Alberta, Canada: Department of Computing Science, University of Alberta, 1990

[21] Bradshaw, D. P. 1993. Composite multidatabase system concurrency control and recovery. In *Proceedings of the 1993 Conference of the Centre For Advanced Studies on Collaborative Research: Distributed Computing - Volume 2* (Toronto, Ontario, Canada, October 24 - 28, 1993). A. Gawman, E. Kidd, and P. Larson, Eds. IBM Centre for Advanced Studies Conference. IBM Press, 895-909.

[22] Dimitrios Georgakopoulos , Marek Rusinkiewicz , Amit P. Sheth, On Serializability of Multidatabase Transactions Through Forced Local Conflicts, Proceedings of the Seventh International Conference on Data Engineering, p.314-323, April 08-12, 1991

[23] Hector Garcia-Molina, Using semantic knowledge for transaction processing in a distributed database, ACM Transactions on Database Systems (TODS), v.8 n.2, p.186-213, June 1983

[24] Hector Garcia-Molina , Kenneth Salem, Sagas, Proceedings of the 1987 ACM SIGMOD international conference on Management of data, p.249-259, May 27-29, 1987, San Francisco, California, United States

[25] Henry F. Korth , Eliezer levy , Abraham Silberschatz, A formal approach to recovery by compensating transactions, Proceedings of the sixteenth international conference on Very large databases, p.95-106, September 1990, Brisbane, Australia

[26] Eliezer Levy , Henry F. Korth , Abraham Silberschatz, An optimistic commit protocol for distributed transaction management, Proceedings of the 1991 ACM SIGMOD international conference on Management of data, p.88-97, May 29-31, 1991, Denver, Colorado, United States

[27] Y. Breitbart and A. Silberschatz, "Strong Recoverability in Multidatabase Systems," Proc. Second Int'l Workshop Research Issues on Data Eng.: Transaction and Query Processing, pp. 170-175, Tempe, Ariz., 1992.

[28] Meichun Hsu , Abraham Silberschatz, Unilateral Commit: A New Paradigm for Reliable Distributed Transaction Processing, Proceedings of the Seventh International Conference on Data Engineering, p.286-293, April 08-12, 1991

[29] S. Mehrotra R. Rastogi H.F. Korth and A. Silberschatz, "A Transaction Model for Multidatabase Systems," Proc. Int'l Conf Distributed Computing Systems, June 1992

[30] Recovery in Multidatabase Systems. Angelo Brayner. Federal University of Ceara. brayner@lia.ufc.br. Theo Harder. University of Kaiserslautern ...

[31] Barkaoui, K. and Benamara, R. 2003. On Concurrency Control in Multidatabase Systems with an Extended Transaction Model. *J. Supercomput.* 24, 2 (Feb. 2003), 193-202

[32] Breitbart, Y., Silberschatz, A., and Thompson, G. R. 1990. Reliable transaction management in a multidatabase system. In *Proceedings of the 1990 ACM SIGMOD international Conference on Management of Data* (Atlantic City, New Jersey, United States, May 23 - 26, 1990). SIGMOD '90. ACM Press, New York, NY, 215-224

[33] Y. Breitbart, A. Silberschatz, and G. R. Thompson. *An approach to recovery management in a multidatabase system*. The VLDB Journal, 1(1):1, July

[34] Y. Breibart, W. Litwin and A. Silberschatz, `*Deadlock problems in a multidatabase environment*', in Proc. COMPCON, 1991

APPENDIX A

This appendix holds the procedures for TM

```
/* Parses the global query and sends subtransactions to individual agents*/

public Hashtable parse(String query) {

    String tt_query = query.toUpperCase().trim();
    int siteIndex = tt_query.indexOf(":");
    String sitename = tt_query.substring(0,siteIndex).trim();
    int tIndex = tt_query.indexOf("\\");
    String tname = tt_query.substring(siteIndex+1,tIndex).trim();
    String strnname = sitename +"_"+ tname;
    String t_query = tt_query.substring(tIndex+2).trim();
    new_query = t_query;


/* If the query is a UPDATE statement */

    if (t_query.startsWith("UPDATE ")) {
       String temp_query = t_query;
        int dtindex1 = temp_query.indexOf("UPDATE ")+7;
        String droptemp = temp_query.substring(dtindex1).trim();
        int dtindex2 = droptemp.indexOf(" SET ");
        String db_and_tables = droptemp.substring(0,dtindex2).trim();
        //System.out.println("db_and_tables ="+db_and_tables);
        alltables=db_and_tables;
        while (db_and_tables.indexOf(",")>0)
        {
                String db_and_table =
              db_and_tables.substring(0,db_and_tables.indexOf(",")).trim();
                db_and_tables =
              db_and_tables.substring(db_and_tables.indexOf(",")+1).trim();
                map_ddl(db_and_table,strnname);
              //System.out.println("db_and_table in loop ="+db_and_table);
        }
                map_ddl(db_and_tables,strnname);
              //System.out.println("db_and_tables in loop ="+db_and_tables);
    } //if

  /* If the query is a SELECT statement */

    while (t_query.startsWith("SELECT ")) {
       String temp_query = t_query;
        int fromIndex=temp_query.indexOf("FROM")+4;
        //System.out.println("from index = "+fromIndex);
        int whereIndex=-1;
        try{
              whereIndex = temp_query.indexOf("WHERE");
           }catch(Exception e){}

        if (whereIndex<0){
              temp_query = temp_query.substring(fromIndex).trim();
                //System.out.println("temp_query = "+  temp_query );
        }
        else {
               temp_query =
             temp_query.substring(fromIndex,whereIndex).trim();
             //System.out.println("temp_query = "+  temp_query );
       }

       while (temp_query.indexOf(",") > 0)
          {
```

```
                String db_and_table =
            temp_query.substring(0,temp_query.indexOf(",")).trim();

                if (db_and_table.indexOf(" ")>0)
                    db_and_table =
                    db_and_table.substring(0,db_and_table.indexOf(" "));
                    String s=map_dt(db_and_table,strnname);
                    new_query = replaceAll(new_query,db_and_table,s);
                    temp_query =
                    temp_query.substring(temp_query.indexOf(",")+1).trim();
         }

        if (temp_query.indexOf(")")>=0)
            temp_query = temp_query.substring(0,temp_query.indexOf(")"));
      if (temp_query.indexOf(" ")>0)
            temp_query =
            temp_query.substring(0,temp_query.indexOf(" "));
            //System.out.println("temp_query = "+  temp_query );

        String s = map_dt(temp_query,strnname);
        //System.out.println("s = "+ s);
        new_query = replaceAll(new_query,temp_query,s);
        //System.out.println("new_query = "+  new_query );
        t_query = t_query.substring(t_query.indexOf("SELECT ")+7);
        try{
        t_query = t_query.substring(t_query.indexOf("SELECT "));
           }catch(Exception e){}

  } //while
   return ht;

}//end of parse()
/******************* returns the new query
*************************************/
String getModifiedQuery() {
    return new_query;
}
/*************************map_ddl*****************/

String map_ddl(String db_and_table,String strnname) {
    String temp_new_query = new_query;
    String db="";
    String table="";
    //System.out.println("map_ddl(db_and_table) for"+db_and_table);

     if (db_and_table.indexOf(".")>0) {
       db =
       db_and_table.substring(0,db_and_table.indexOf(".")).trim();
        table =
       db_and_table.substring(db_and_table.indexOf(".")+1).trim();
        temp_new_query =
       replaceAll(temp_new_query,alltables,table);

    } else {
       db = masterDB;
        table = db_and_table.trim();
       //System.out.println("alltables"+alltables);
        temp_new_query =
       replaceAll(temp_new_query,alltables,table);
   }

   //System.out.println(db+":"+temp_new_query);

    Hashtable dbases = new Hashtable();
    Hashtable tables = new Hashtable();

    //System.out.println("table="+table);
    //System.out.println("db="+db);
    //System.out.println("temp_new_query="+temp_new_query);

    if (!ht.containsKey(strnname)) {
       tables.put(table,temp_new_query);
```

```
        dbases.put(db,tables);
      ht.put(strnname,dbases);
      }else {
            dbases = (Hashtable) ht.get(strnname);
            if(!dbases.containsKey(db)) {
                  tables.put(table,temp_new_query);
                  dbases.put(db,tables);
                  ht.put(strnname,dbases);
                  } else {
                        tables = (Hashtable) dbases.get(db);

                        if(!tables.containsKey(table))  {
                              tables.put(table,temp_new_query);
                              } else {
                                    tables.put(table,
(String)tables.get(table)+"|"+temp_new_query);
                              }
            dbases.put(db,tables);
               ht.put(strnname,dbases);

            }
    }

    return table;
}
/**************** identifies the dbname and table name
**************************/
String map_dt(String db_and_table,String strnname) {

    String db="";
    String table="";
    String temp_table="";

    if (db_and_table.indexOf(" ")>0)
       db_and_table = db_and_table.substring(0,db_and_table.indexOf(" "));

    if (db_and_table.indexOf(".")>0) {
       db =
       db_and_table.substring(0,db_and_table.indexOf(".")).trim();
        table =
       db_and_table.substring(db_and_table.indexOf(".")+1).trim();

       } else {
             db = masterDB;
             table = db_and_table.trim();
       }

    String temp_query = "select * from "+table;
    //if (db.equals(""))
    temp_table = table;
       //else
       //    temp_table = db+"_"+table;

    Hashtable dbases = new Hashtable();
    Hashtable tables = new Hashtable();

    if (!ht.containsKey(strnname)) {
       tables.put(temp_table,temp_query);
        dbases.put(db,tables);
       ht.put(strnname,dbases);
       } else {
             dbases = (Hashtable) ht.get(strnname);

             if(!dbases.containsKey(db)) {
                   tables.put(table,temp_query);
                       dbases.put(db,tables);
                       ht.put(strnname,dbases);
                       } else {
                          tables = (Hashtable) dbases.get(db);

                                if(!tables.containsKey(table))  {
                                        tables.put(table,temp_query);
```

```
                                } else {
                                        tables.put(table,
(String)tables.get(table)+"|"+temp_query);
                                }

                                dbases.put(db,tables);
                                ht.put(strnname,dbases);
                        }

             }//if end

        //System.out.println(db+":"+temp_table+":"+temp_query);
return table;
}

/************ replaces s2 with s3 in string s1 *************************/
String replaceAll( String s1, String s2, String s3) {

    String ans="";
    while (s1.indexOf(s2)>=0) {
       String tb=s1.substring(0,s1.indexOf(s2));
        String ta=s1.substring(s1.indexOf(s2)+s2.length());
        ans=ans+tb+s3;
        s1=ta;
    }

    ans=ans+s1;
    return ans;
}//end replaceall

}//end of querypartser


/****************************CreateSockets Class******************************/
class CreateSockets {
       Socket[] socks;
       ObjectOutputStream oos1;

       public void initialize(int size) {
       try {
        socks = new Socket[size];
       }catch(Exception e) {
        System.out.println("Exception in initialize method:"+e.getMessage());
       }
       }//end of initialize()

       public void create(int i,int port,Hashtable inHashTable) {

try {
        System.out.println(" ");
        System.out.println("****Sockets Creation***************");
        try {
                socks[i] = new Socket(InetAddress.getLocalHost (), port);
                System.out.println("sock "+i+" created on port "+port);
                System.out.println("hashtable created"+inHashTable);
                oos1 = new ObjectOutputStream(socks[i].getOutputStream());
                oos1.writeObject(inHashTable);
                oos1.flush();
             int j = i+1;
                System.out.println("hash table sent to agent"+j);

        }catch (Exception e){
        socks[i] = null;
        }
        System.out.println("******End of Sockets Creation*************");
        System.out.println(" ");
        System.out.println(" ");

}catch(Exception e) {
        System.out.println("Exception in sendgcommit method:"+e.getMessage());
}
}//end of create()
```

```
public String sendmessages(int s) {

        String vote = "";
        String ms1;

try {

    //    BufferedReader inFromUser =
      //          new BufferedReader(new InputStreamReader(System.in));

        DataOutputStream outToAgent1 =
                new DataOutputStream(socks[s].getOutputStream());

        BufferedReader ReadVote1 =
                new BufferedReader(new
                        InputStreamReader(socks[s].getInputStream()));
/*
        //read message1 from user
        System.out.println("Enter message to Agent"+s);
        ms1 =  inFromUser.readLine();
        outToAgent1.writeBytes(ms1 + '\n');
       outToAgent1.flush();
  */
        //read vote from Agent0
        vote = ReadVote1.readLine();
        System.out.println("Vote From Agent"+s+":"+vote);

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

}catch(Exception e) {
        System.out.println("Exception in sendgcommit method:"+e.getMessage());
}
        return vote;
}//end of sendmessages()

public void sendcommit(int s,String globalcommit) {

try {
         DataOutputStream WriteGcommit1 =
                new DataOutputStream(socks[s].getOutputStream());

        //send gcommit to all agents
        WriteGcommit1.writeBytes(globalcommit + '\n');
        WriteGcommit1.flush();
        System.out.println("globalcommit sent to Agent"+s);
}catch(Exception e) {
        System.out.println("Exception in sendgcommit method:"+e.getMessage());
}
}//end of sendcommit()

String rcvAck(int s) {
       String ack = " ";
try {
        BufferedReader ReadAck =
                new BufferedReader(new
                        InputStreamReader(socks[s].getInputStream()));
        ack = ReadAck.readLine();
        System.out.println("Acknowledgement From Agent"+s+":"+ack);

}catch(Exception e) {
        System.out.println("Exception in rcvAck method:"+e.getMessage());
}
 return ack;
}//end of rcvAck()

}//end of CreateSocket class

/*************************End of ConnectionToClient Class******************/
/*************************Manager Class and main()********************/
```

```
public class dmgr {
       public static void main(String args[])throws SQLException,IOException {

                int i=0,no_of_queries = 0;
                String[] file_lines = new String[50];
                String[] queries = new String[50];

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

        System.out.println("*****************Print Input
File*************************************");

        if (args.length == 1) {
        try {
                FileInputStream fstream = new FileInputStream(args[0]);
                DataInputStream in = new DataInputStream(fstream);
                while (in.available() !=0) {
                // Print file line to screen
                file_lines[i] = in.readLine();
             queries[i] = file_lines[i];
                System.out.println (queries[i]);
                i++;
             no_of_queries = i;
             }//end of file print while

             in.close();
             System.out.println("*****************End Of Input
File*************************************");
             System.out.println(" ");
             System.out.println(" ");


       }  catch (Exception e1){
             System.err.println("File input error");
        }
        } else//end of if(args)
        System.out.println("Invalid parameters");

       Connection conn = null;

//Establish jdbc connection


  try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
      } catch (ClassNotFoundException e2) {
        System.out.println("Could not load the driver");
      }

try {

    conn = DriverManager.getConnection (
         "jdbc:oracle:thin:@tinman.cs.gsu.edu:1521:tinman",
         "smadiraju1","smadiraju1");
  } catch (SQLException e3) {
     System.out.println("Error connecting to Oracle:"+e3.getMessage());
      return;
  }

        String sites = "select * from sites";
        Statement stmt1 = conn.createStatement();
        ResultSet rset1 = null;

try {

        rset1 = stmt1.executeQuery(sites);
        System.out.println("******************Sites
Table************************************");
        System.out.println(" Siteid Sitename Url Portno Tables");
        while (rset1.next ()) {
             System.out.print(rset1.getString(1)+ "    " + rset1.getString(2));
```

```
            System.out.print("    " +rset1.getString(3)+ "   " +
rset1.getString(4));
            System.out.println("   " +rset1.getString(5));
        }
        System.out.println(" ");
        System.out.println("*******************End Of Sites
Table************************************");
        System.out.println(" ");
        System.out.println(" ");
 }catch (SQLException e4) {
     System.out.println("Error in  getting table contents2:"+e4.getMessage());
      return;
  }

      QueryParser parser = new QueryParser();

      Hashtable outsideHashTable =  new Hashtable();
      Enumeration enum1;
      //no_of_queries =no_of_queries - 1;
      System.out.println("no of queries="+no_of_queries);

      String[] sitenames = new String[10];
      Set stnames = new HashSet();

      System.out.println(" ");
       System.out.println(" ");
      System.out.println("*******************Queries sent to
parser***********************************");
      for(int q=0;q<no_of_queries;q++){

            System.out.println("query"+q+": "+queries[q]);
            System.out.println("-----");

            outsideHashTable = parser.parse(queries[q]);
      }//for end no_of_queries

      System.out.println("********************End of
Queriese************************************");

      //Entite hashtable from parser()
      System.out.println("********************Entire
Hashtablee**********************************");
      System.out.println(" ");
       System.out.println(" ");
      System.out.println ("Entire hashtable from parser:"+outsideHashTable);
       System.out.println(" ");
       System.out.println(" ");
       System.out.println("********************End of
Hashtable***********************************");

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

      enum1 = outsideHashTable.keys();

      Hashtable inHashTable =  new Hashtable();
      while(enum1.hasMoreElements()) {
            String st_tid = (String) enum1.nextElement();
            //inHashTable = (Hashtable) outsideHashTable.get(st_tid);
               //System.out.println (inHashTable);
            int index = st_tid.indexOf("_");
            //System.out.println(" index "+ index );
            String st = st_tid.substring(0,index).trim();
            //add sitenames into a set to avoid duplicates
            stnames.add(st);
            } //end of enum1
      sitenames = (String[])stnames.toArray(new String[stnames.size()]);
      int size = stnames.size();
      int eachsite = 0;
/*    //print sitenames
      for(eachsite=0;eachsite< size;eachsite++){
             System.out.println("sitename: "+sitenames[eachsite]);
```

```
        }
*/
        //Create an instance of the CreateSockets Class
        CreateSockets newsock = new CreateSockets();
         newsock.initialize(size);
        String[] votes = new String[10];

        for(eachsite=0;eachsite< size;eachsite++){

        Statement stmt2 = conn.createStatement();
        System.out.println(" ");
         System.out.println("****************eachsite
loop****************************************");
         System.out.println(" ");

        int portnum = 0;

         for(eachsite=0;eachsite< size;eachsite++){

               ResultSet rset2 = null;
                  String ptno = "select distinct(portno) from sites where sitename
='"+sitenames[eachsite]+"'";
                  //System.out.println("ptno:"+ptno);
                  rset2 = stmt2.executeQuery(ptno);

                  while (rset2.next ()) {
                  portnum = rset2.getInt(1);
                  }

               System.out.println("portnum:"+portnum);

                  //System.out.println("stnames for which connection is to be
established = "+sitenames[eachsite]);

               String site = sitenames[eachsite];
               System.out.println ("SITE:"+site);
               enum1 = outsideHashTable.keys();
               Hashtable siteHashTable =  new Hashtable();
               while(enum1.hasMoreElements()) {

                      String st_tid = (String) enum1.nextElement();
                      int index = st_tid.indexOf("_");
                      //System.out.println(" index "+ index );
                      String st = st_tid.substring(0,index).trim();
                      //System.out.println ("st:"+st);
                      if (st.equals(site)) {
                             inHashTable = (Hashtable)
outsideHashTable.get(st_tid);
                             //System.out.println("inHashtable"+inHashTable);
                             siteHashTable.put(st_tid,inHashTable);
                             //System.out.println
("siteHashtable="+siteHashTable);
                      } else {
                             continue;
                             }

               }

               System.out.println("siteHashTable="+siteHashTable);

               //Establish connection to port and send the hashtable

               newsock.create(eachsite,portnum,siteHashTable);
               siteHashTable.clear();
               votes[eachsite] = newsock.sendmessages(eachsite);
               System.out.println(" ");
               System.out.println(" ");
        }

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

```
        //Print Votes from Agents
        for(int site=0;site<size;site++){
              System.out.println("votes from connect:"+ votes[site]);
        }//end of for votes
         int  no_Of_Agents = size;
        System.out.println("no_Of_Agents:"+no_Of_Agents);
        String[] decision = new String[10];
         String[] gcommit = new String[10];
        int count = 0;
        for(i = 0;i < no_Of_Agents;i++) {
                 if (votes[i].equals("commit")) {
              count++;
              //if no vote is recieved set default vote to abort
              } else  if (votes[i].equals(" ")) {
                    votes[i] = "abort";
              }
        }//end of for
        System.out.println("Number of Commits = "+count);
        String globalcommit = " ";
        if (count == no_Of_Agents) {
        globalcommit = "commit";
        System.out.println("gcommit="+globalcommit);
        } else {
              globalcommit = "abort";
              System.out.println("gcommit="+globalcommit);
        }

        //Send globalcommit to all Agents
        System.out.println(" ");
        System.out.println(" ");
        //System.out.println("sendgcommit called");
        for(int s=0;s<size;s++){
        //System.out.println("sendgcommit for socks"+s);
         newsock.sendcommit(s,globalcommit);
        }

        String[] acks = new String[10];

        System.out.println("rcvAck called");
         for(int a=0;a<size;a++){
         System.out.print("Acknowledgement from socks"+a);
         acks[a] = newsock.rcvAck(a);
         System.out.println(acks[a]);
          }


        System.out.println(" ");
         System.out.println(" ");
}//end of main
}//end of dmgr class
```

APPENDIX B

This appendix holds the procedures for Agents

```
   public class Agent1 {
      public static void main (String args[])
         throws IOException,SQLException {

try {
       Connection con = null;

       //Establish jdbc connection
  try {
         Class.forName("oracle.jdbc.driver.OracleDriver");
       } catch (ClassNotFoundException e) {
         System.out.println("Could not load the driver");
       }

try {

    con = DriverManager.getConnection (
         "jdbc:oracle:thin:@tinman.cs.gsu.edu:1521:tinman",
         "site1","site1");
  } catch (SQLException e1) {
     System.out.println("Error connecting to Oracle:"+e1.getMessage());
      return;
  }
Statement stmt1 = con.createStatement();
/****************************/
class createConnection {
       Connection[] conn;
       public void initialize() {

try {
       conn = new Connection[5];
        conn[1] = null;
        conn[2] = null;
        conn[3] = null;
       }catch(Exception e) { }
}//end of initialize()


       public void  connectToDatabase(int cc,String dbname) {

try {

       int cn = cc;
       String uid = "";
        String pwd = "";

       if(dbname.equals("DB2")) {
              uid = "site2";
              pwd = "site2";
       } else if(dbname.equals("DB3")) {
              uid = "site3";
                 pwd = "site3";
       }else {
              uid = "site4";
                 pwd = "site4";
       }

    conn[cn] = DriverManager.getConnection (
         "jdbc:oracle:thin:@tinman.cs.gsu.edu:1521:tinman",
         uid,pwd);
       System.out.println ("connection established to conn"+cn);
```

```
  } catch (SQLException e1) {
     System.out.println("Error connecting to Oracle:"+e1.getMessage());
      return;
  }

}//end of connectToDatabase()

public int lockTables(int c,String tablename,String query) {
int upd = 0;
try {
        Statement stmt2 = conn[c].createStatement();
       //conn[c].setAutoCommit(false);
       System.out.println ("Locking Table...: "+tablename);
        stmt2.executeUpdate("LOCK TABLE "+ tablename + " IN EXCLUSIVE MODE
NOWAIT");
       conn[c].setAutoCommit(false);
       upd = stmt2.executeUpdate(query);
} catch(Exception e) {
       System.out.println ("Exception in Locking Tables"+e.getMessage());
       //try {conn[c].setAutoCommit(false);}catch(Exception e8){}
       }
return upd;
}//end of lockTables()

public void callCommit() {

try {
       conn[1].commit();
       conn[2].commit();
       conn[3].commit();
}catch(Exception e) {
       //System.out.println ("Exception in callCommit()"+e.getMessage());
}

}//end of callCommit()

public void callRollback() {
try {
         conn[1].rollback();
        conn[2].rollback();
        conn[3].rollback();

}catch(Exception e) {
        //System.out.println ("Exception in callRollback()"+e.getMessage());
}

}//end of callRollback()

public void setCommit(){
try {
        conn[1].setAutoCommit(true);
        conn[2].setAutoCommit(true);
        conn[3].setAutoCommit(true);
}catch(Exception e) {
        //System.out.println ("Exception in setCommit()"+e.getMessage());
}

}//end of setCommit()

/*
public int execute(int c,String query) {
 int upd = 0;

try {
        Statement stmt3 = conn[c].createStatement();
        System.out.println ("executing update");
        upd = stmt3.executeUpdate(query);
} catch(Exception e) {
        //System.out.println ("Exception in executing queries"+e.getMessage());
        }

return upd;
```

```
}//end of execute()
*/
}//end of createConnections class

       String   preparemsg = " ",vote = " ",response = " ";
       int port = 6010;
       int i=0,j=0;

       Hashtable  ht;
       ObjectInputStream ois;

       ServerSocket s2 = new ServerSocket (port);
       Socket sock2;
       System.out.println (" ");
       System.out.println (" ");
       System.out.println ("*******************************************");
       System.out.println ("ag1 starting ...to communicate with TCmanager on
port"+port);
       sock2 = s2.accept ();

       try {
              sock2.setSoTimeout(20000);
              }catch(SocketException e) {
              e.printStackTrace();
       }

       System.out.println("client accessed");

       ois = new ObjectInputStream(sock2.getInputStream());
       ht = (Hashtable)ois.readObject();
       //System.out.println(ht);

       //sort the Hashtable
       Vector v = new Vector(ht.keySet());
       Iterator it = ht.keySet().iterator();
       Collections.sort(v);
       it = v.iterator();
       Hashtable sortHashTable =  new Hashtable();
       Hashtable dbases =  new Hashtable();
       Hashtable tables =  new Hashtable();
       Hashtable tablelocks =  new Hashtable();
       Hashtable dbqueries =  new Hashtable();
       String[] queries = new String[50];
       int q = 0,c = 0,nrows = 0,count = 0;
       int no_of_databases = 0;
        Enumeration enum1;
       Enumeration enum2;
       Set dbnames = new HashSet();
       String[] con_dbs = new String[10];
       int  no_of_updates = 0;
       createConnection create = new createConnection();
       create.initialize();
       while (it.hasNext()) {
                     String site =  (String)it.next();
              sortHashTable = (Hashtable) ht.get(site);
                     System.out.println(site);
              System.out.println("-------");
              System.out.println("");
              enum1 =  sortHashTable.keys();
              while(enum1.hasMoreElements()) {
                     String db = (String) enum1.nextElement();
                     System.out.print("db:"+db);
       try {
                if(!db.equals("DB1")) {

                     if(!dbnames.isEmpty()) {
                            //connect to databases

                            if (!dbnames.contains(db)) {
                                   dbnames.add(db);
                                   c++;
```

```
                                System.out.println("Connect to "+db);
                                create.connectToDatabase(c,db);
                        } else {
                                System.out.println("Connection already exists
for"+db);
                        }//end if contains
                } else {
                        dbnames.add(db);
                        c++;
                        System.out.println("Connect to "+db);
                        create.connectToDatabase(c,db);
                }//end if empty
            } else {
                    System.out.println("default site"+db);

            }//end of if DB1

        }catch(Exception e) { }

                con_dbs = (String[])dbnames.toArray(new
String[dbnames.size()]);
                no_of_databases = dbnames.size();
                dbases = (Hashtable) sortHashTable.get(db);
                //System.out.println(dbases);

                enum2 =  dbases.keys();
                while(enum2.hasMoreElements()) {
                        String tab = (String) enum2.nextElement();
                        System.out.print("Table:"+tab);
                        queries[q] = (String) dbases.get(tab);
                        System.out.println("queries"+ queries[q]);
                try {
                        if(!dbqueries.containsKey(db)) {
                                                dbqueries.put(db,queries[q]);
                                        } else {

dbqueries.put(db,(String)dbqueries.get(db)+"|"+queries[q]);
                                        }//end if
                    }catch(Exception e) {}

                        String t_query = queries[q];
                        q++;

                try {
                        //if UPDATE query add to tablelocks
                         if (t_query.startsWith("UPDATE")) {
                               no_of_updates++;
                              if(!db.equals("DB1")) {
                               //System.out.println("update");
                               if(!tablelocks.containsKey(db)) {
                                       tablelocks.put(db,tab);
                                       nrows =
create.lockTables(c,tab,t_query);
                                       //nrows = create.execute(c,t_query);
                                       System.out.println("nrows =
"+nrows+t_query);
                                       if(nrows > 0) {
                                           count++;
                                           nrows = 0;
                                       } else {
                                         count = 0;
                                         nrows = 0;
                                       }//end nrows
                                       //System.out.println("count = "+count);
                                 } else {

       tablelocks.put(db,(String)tablelocks.get(db)+"|"+tab);
                                       nrows =
create.lockTables(c,tab,t_query);
                                       //nrows = create.execute(c,t_query);
                                       System.out.println("nrows =
"+nrows+t_query);
```

```
                                        if(nrows > 0) {
                                                   count++;
                                           nrows = 0;
                                                } else {
                                                  count = 0;
                                         nrows = 0;
                                                }//end nrows

                                  }//end if
                               } else {
                                 System.out.println ("Locking Table...: "+tab);
                                       stmt1.executeUpdate("LOCK TABLE "+ tab +
" IN EXCLUSIVE MODE NOWAIT");
                                 con.setAutoCommit(false);
                                 nrows = stmt1.executeUpdate(t_query);
                                 System.out.println("nrows default =
"+nrows+t_query);
                                  if(nrows > 0) {
                                                   count++;
                                           nrows = 0;
                                                } else {
                                                  count = 0;
                                         nrows = 0;
                                                }//end nrows

                                }//end of if DB1

                           } else {
                                  continue; }
                    }catch(Exception e) {
                                     System.out.println ("Exception in
locking"+e.getMessage());
                                     System.out.println("nrows in exception =
"+nrows);
                                       if(nrows > 0) {
                                                   count++;
                                                   nrows = 0;
                                                } else {
                                                  count = 0;
                                                  nrows = 0;
                                                }//end nrows

                     }

                    }//while enum2

             }//while enum1

      }//while it
      //end of sorting hashtable

      System.out.println("count = "+count);
      System.out.println("no_of_updates="+no_of_updates);
      if(no_of_updates == 0) {
             vote = "commit";
      } else {
             if(count == no_of_updates) {
                vote = "commit";
                      } else {
                        vote = "abort";
             }//end of if
      }
       System.out.println("vote="+vote);
       BufferedReader inFromClient2 = new BufferedReader(new
                       InputStreamReader(sock2.getInputStream()));
       DataOutputStream outToClient2 =
                new DataOutputStream(sock2.getOutputStream());

      response = vote + "\n" ;
       outToClient2.writeBytes(response);
      outToClient2.flush();
```

```
        String gcomm;
         gcomm = inFromClient2.readLine();
         System.out.println("gcommit recieved from managre"+gcomm);

         if(gcomm.equals("commit")) {
               create.callCommit();
               con.commit();
                  } else if(gcomm.equals("abort")) {
                      create.callRollback();
                      con.rollback();
                  } else {
                      create.callRollback();
                          con.rollback();
               }


        create.setCommit();
        con.setAutoCommit(true);

        System.out.println("sending Acknowledgement of gcommit message");
        String ack = gcomm.toUpperCase().trim();
        outToClient2.writeBytes(ack);
        outToClient2.flush();
        System.out.println("");
        System.out.println("");
        int no_of_queries = q;
        System.out.println("no_of_queries"+ no_of_queries);

        System.out.println("");
         System.out.println("");
        System.out.println("tablelocks:"+ tablelocks);

        System.out.println("");
         System.out.println("");
        System.out.println("dbqueries:"+dbqueries);
        System.out.println("");
         System.out.println("");
        for(int d = 0;d<no_of_databases;d++) {
               System.out.println("con_dbs"+con_dbs[d]);
        }
        System.out.println("");
         System.out.println("");
         sock2.close();
         s2.close();

}catch (Exception e) {
     System.out.println("Error creating objectstream:"+e.getMessage());
     e.printStackTrace();
      //return;
  }

  }
   }
```