ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 4, Issue 3, pp: (233-244), Month: July - September 2016, Available at: www.researchpublish.com

AN ALGORITHM FOR THE DISTRIBUTED TWO-PHASE COMMIT PROTOCOL

Christos Papanastasiou

Technological Educational Institution Stereas Elladas, Department of Electrical Engineering, 35100 Lamia, GREECE

Abstract: **Many different schemes have been developed and used in distributed database systems to coordinate the execution of subtransactions of a global transaction. On this paper I present an algorithm for the distributed two-phase commit protocol to give a chance to some transactions to complete when they are ready to abort. First I count the number of transactions who have decided to abort. If this count is less than 35 percent of the total participants, then for only these transactions I suggest to prepare again for a completion. In this way, we can ensure that a 30 percent of transactions who had decided to abort, can complete their execution.**

Keywords: **two-phase commit protocol, transactions, transaction files, distributed database.**

1. INTRODUCTION

A distributed database is a single logical database that is spread physically across computers in multiple locations that are connected by a data communication network.

The network must allow the users to share the data in order that a user or program at a location A to be able to access and maybe update the data that are in another location, location B. The sites of a distributed system may be spread over a large geographical or over a small area. Data can be replicated over a network using horizontal and vertical fragmentation. The database share the same problems of access control and transaction management, such as user concurrent access control and deadlock detection and resolution.

Distributed database systems (DDBS) are systems that have their data distributed and replicated over several locations; unlike the centralized data base system (CDBS), where only one copy of the data is stored. Data may be replicated over a network using horizontal and vertical fragmentation

Access control and transaction management in Distributed Database Ssystems have to follow different rules to monitor data retrieval and update to distributed and replicated databases. A Database Management Systems (DBMS) can apply the two-phase commit technique to maintain a consistent state for the databases used. The objective of this paper is to explain an algorithm to be used in The Distributed Two-Phase Commit Protocol.

There are advantages, disadvantages, and failures in Distributed Database Systems. (Connolly et al., 1997)

1.1. Advantages of a Distributed Data Base System:

Many organizations are geographically dispersed, which means that in this case a DDBS fits the organizational structure better than traditional centralized DBS. Each location will have its local data as well as the ability to get needed data from other locations throug a communication network. If a failure of one of the servers happens at one site it will have the distributed database system to be inaccessible. In addition, if any data is required from a site which is in failure, these data can be retrieved from other locations who contain the replicated data.

The performance of the system is improved since several machines take care of distributing the load of the CPU and the I/O.

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 4, Issue 3, pp: (233-244), Month: July - September 2016, Available at: www.researchpublish.com

1.2 Disadvantages of Distributed DBS:

Distributed Database Systems have some disadvantages. A distributed system usually exhibits more complexity and cost more than a centralized one. This is because the hardware and software involved need to maintain a reliable and an efficient system. Replication and data retrieval from all sites should be transparent to the user. The cost of maintaining the system is considerable since technicians and experts are required at every site.

Costs more in terms of software cost compared to a centralized system. Additional software might be needed in most of the cases over a centralized system. It costs many messages to be shared between sites to complete a distributed transaction. Data integrity becomes complex because too much network resources may be used.

1.3 Failures in Distributed DBS:

Several types of failures may occur in a distributed database system:

- **Transaction Failures:** When a transaction fails, it aborts. Thereby, the database must be restored to the state it was in before the transaction started. Transactions may fail for several reasons. Some failures may be due to deadlock situations or concurrency control algorithms.
- **Site Failures:** Site failures are usually due to software or hardware failures. These failures result in the loss of the main memory contents. In distributed database, site failures are of two types:

1. *Total Failure* where all the sites of a distributed system fail,
2. *Partial Failure* where only some of the sites of a distributed system fail.

- **Media Failures:** Such failures refer to the failure of secondary storage devices. The failure itself may be due to head crashes, or controller failure. In these cases, the media failures result in the inaccessibility of part or the entire database stored on such secondary storage.

- **Communication Failures:** Communication failures, as the name implies, are failures in the communication system between two or more sites. This will lead to network partitioning where each site, or several sites grouped together, operates independently. As such, messages from one site won't reach the other sites and will therefore be lost. The reliability protocols then utilize a timeout mechanism in order to detect undelivered messages. A message is undelivered if the sender doesn't receive an acknowledgment. The failure of a communication network to deliver messages is known as performance failure, [7].

2. TRANSACTIONS

A transaction is a unit of program execution that accesses and possibly updates various data objects. Usually a transaction is initiated by a user program written in a high-level language (C++, Java, SQL), where it is delimited by statements or function calls [8].

To ensure integrity of data, we require that the database system maintain the following properties of the transactions: [8].

- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation:** Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist even if there are system failures.

These properties are often called **ACID properties**, the acronym is derived from the first letter of each of the four properties.

To gain a better understanding of the ACID properties and the need for them, consider a simplified banking system consisting of several accounts and a set of transactions that access and update those accounts. For the time being, we assume that the database permanently resides on disk but that some portion of it is temporarily residing in main memory.

Transactions access data using two operations:

- Read (O), which transfers the data object O from the database to a local buffer belonging to the transaction that executed the read operation.
- Write (O), which transfers the data object O from the local buffer of the transaction that executed the write back to the database.

In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk. The write operation may be temporarily stored in memory and executed on the disk later. For now, however, we will assume that the write operation updates the database immediately. [8]

Let's suppose that T_i is a transaction that transfers 20 euros from account A to account B. This transaction can be defined as :

```
Ti :    read(A)
        A=A-20
        write(A)
        read(B)
        B=B+20
        Write(B)
```

Let's consider now each of ACID properties:

- **Consistency:** The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction. It can be verified easily that if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.
- **Atomicity:** Suppose that just before the execution of transaction T_i, the values of accounts A and B are 3000 euros and 5000 euros respectively. If we suppose that during the execution of transaction T_i a failure occurs that prevents T_i from completing its execution successfully. Failure might happen because of power failures, hardware failures and software errors. If we suppose that the failure happened after the write(A) operation but before the write(B) operation, the values of accounts A and B in the database will be 2980 and 5000 euros. The system destroyed 20 euros as a result of this failure which means that the sum A+B is no longer preserved. This is an inconsistent state. We must ensure that such inconsistencies are not visible in a database system. If the atomicity property is present, all actions of the transaction are reflected in the database, or none are. In this case the database system keeps track (on disk) of the old values of any data on which a transaction performs a write and if the transaction does not complete it's execution, the database system restores the old values to make it appear as though the transaction never executed. Ensuring atomicity is the responsibility of the database system itself and is handled by a component called the transaction-management component.
- **Durability:** Once the execution of the transaction completes successfully and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to the transfer of funds. The durability property guarantees that once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. Let's assume now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either:

- 1. The updates carried out by the transaction have been written to disk before the transaction completes.
- 2. Information about the updates carried out by the transaction and written to the disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure. Ensuring durability is the responsibility of a software component of the database system called the recovery-management component.

- **Isolation:** Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state. For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate point and computes A+B, it will observe an inconsistent value. Furthermore, if this second transaction performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed. A way to avoid the problem of concurrently executing transactions is to execute transactions serially, that is one after the other.

3. TRANSACTION STATES

If there are no failures, all transactions will complete successfully. However, a transaction may not always complete its execution successfully. Such a transaction is named **aborted**. If we want to ensure the atomicity property, an aborted transaction does not have to effect on the state of the database. This means that any changes that the aborted transaction made to the database have to be undone. Since the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**. It is part of the responsibility of the recovery scheme to manage transaction aborts. [8]

If a transaction completes its execution successfully we say that is **committed**. A committed transaction that has performed updates, transforms the database into a new consistent state, which must persist even if there is a system failure.

If a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**. For instance, if a transaction added 20 euros to an account, the compensating transaction should subtract 20 euros from this account. It is not always possible to create such a compensating transaction. Thus, the responsibility of writing and executing a compensating transaction depends on the user, and is not handled by the database system.

We have to define exactly what we mean by *successful completion* of a transaction. For this, we can establish a simple abstract transaction model. A transaction must be in one of the following states:

- **Active**, the initial state. The transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after is discovered that normal execution can not proceed.
- **Aborted**, after the transaction has been rolled back and the database has been restored to its prior state to the start of the transaction.
- **Committed**, after successful completion.

A transaction has committed only if it has entered the committed state. We say that a transaction has aborted only if it has entered the aborted state. We say that a transaction is terminated when it has either committed or aborted.

A transaction always starts from the active state. When the transaction executes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory and in this case a hardware failure might prevent its successful completion.

The database system writes enough information to the disk, even if a failure occurs and the updates which have to be performed by the transaction can be re-created when the system restarts after the failure occurs. When the last of this information is written, the transaction enters the committed state.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state and at this point, the system has two options:

- It can **restart** the transaction, but only if the transaction was aborted by some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can **kill** the transaction. It usually does this because of some internal logical error which can only be corrected by rewriting the application program or because the input was bad or because the desired data were not found in the database.

We must be careful when dealing with external writes, such as writes to a terminal or to a printer.

4. TWO-PHASE COMMIT PROTOCOL

The Two-Phase Commit Protocol (2CP) has two types of node to complete its processes: the coordinator and the subordinate, [21]. The coordinator's process is attached to the user application, and communication links are established between the subordinates and the coordinator. The two-Phase Commit protocol goes through two phases. The first phase is a PREPARE phase, which means that the coordinator of the transaction sends a PREPARE message. The second phase is decision-making phase, where the coordinator issues a COMMIT message, if all the nodes can execute the transaction, or an abort message, if at least one subordinate node cannot execute the required transaction.

Two-Phase Commit Protocol (2PC) [19] is said to be blocking because a transaction is blocked due to the coordinator's

failure when the participant is in the ready-to-commit state. On the other hand, Three-Phase Commit Protocol (3PC) is said to be non-blocking [22], [14] but both of these protocols violate the important property of atomicity [9] at the time of multiple site failures for 3PC and single site failure for 2PC.

The 2PC may be carried out with one of the following methods: Centralized 2PC, Linear 2PC, and Distributed 2PC, [7].

4.1 The Centralized Two-Phase Commit Protocol:

In the Centralized 2PC shown in Figure 1 and in Figure 2, communication is done through the coordinator's process only which means that communication between subordinates is not allowed. The coordinator is responsible for transmitting the PREPARE message to the subordinates, and, when the votes of all the subordinates are received and evaluated, the coordinator decides if it will abort or COMMIT. This method has two phases:

1. First Phase: (in Figure 1) In this phase, when a user wants to COMMIT a transaction, the coordinator issues a PREPARE message to all the subordinates, [21]. When a subordinate receives the PREPARE message, it writes a PREPARE log and, if that subordinate is willing to COMMIT, sends a YES VOTE, and enters the PREPARED state; or, it writes an abort record and, if that subordinate is not willing to COMMIT, sends a NO VOTE. A subordinate sending a NO VOTE doesn't need to enter a PREPARED state since it knows that the coordinator will issue an abort. In this case, the NO VOTE acts like a veto in the sense that only one NO VOTE is needed to abort the transaction. The following two rules apply to the coordinator's decision, [7]:

a. If even one participant votes to abort the transaction, the coordinator has to reach a global abort decision.

b. If all the participants vote to COMMIT, the coordinator has to reach a global COMMIT decision.

2. Second Phase: (in Figure 2) After the coordinator gets a vote, it has to relay this vote to the subordinates. If the decision is COMMIT, then the coordinator moves into the committing state and sends a COMMIT message to all the subordinates informing them of the COMMIT. When the subordinates receive the COMMIT message, they move to the committing state and send an acknowledge (ACK) message to the coordinator. When the coordinator receives the ACK messages, it ends the transaction. If, on the other hand, the coordinator reaches an ABORT decision, it sends an ABORT message to all the subordinates. Here, the coordinator doesn't need to send an ABORT message to the subordinate(s) that gave a NO VOTE.

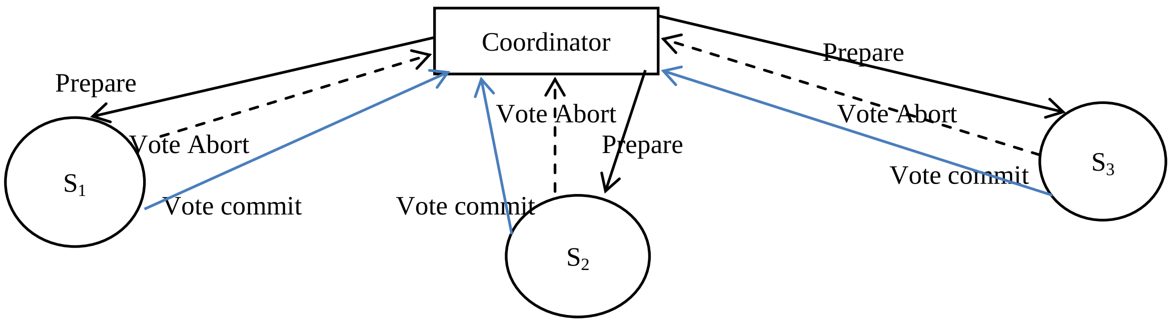

Fig 1: Step One in Centralized Two-Phase Commit Protocol

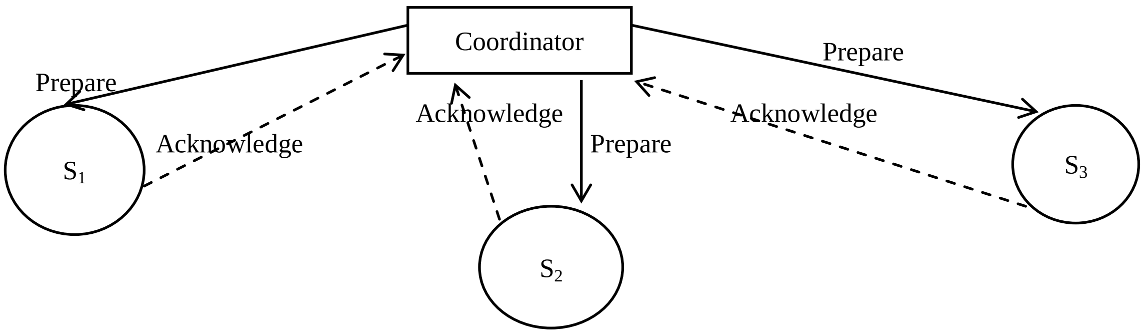

Fig 2: Step Two in Centralized Two-Phase Commit Protocol

4.2 The Linear Two-Phase Commit Protocol:

In the linear 2PC, as we can see in Figures 3 and 4 , subordinates can communicate with each other. The sites are labeled 1 to N, where the coordinator is numbered as site 1. Tthe propagation of the PREPARE message is done serially, this means that the time required to complete the transaction is longer than the time required in centralized or distributed methods. At the end, node N is the one that issues the Global COMMIT. The two phases are discussed below:

First Phase: (Figure 3) The coordinator sends a PREPARE message to participant 2. If participant 2 is not willing to COMMIT, then it sends a VOTE ABORT (VA) to participant 3 and the transaction is aborted at this point. If participant 2, is willing to commit, it sends a VOTE COMMIT (VC) to participant 3 and enters a READY state. Then participant 3 sends its vote till node N is reached and issues its vote.

Second Phase: (Figure 4) Node N issues either a GLOBAL ABORT (GA) or a GLOBAL COMMIT (GC) and sends it to node N-1. Then node N-1 will enter an ABORT or COMMIT state. Node N-1 will send the GA or GC to node N-2 until the final vote to commit or abort reaches the coordinator node.

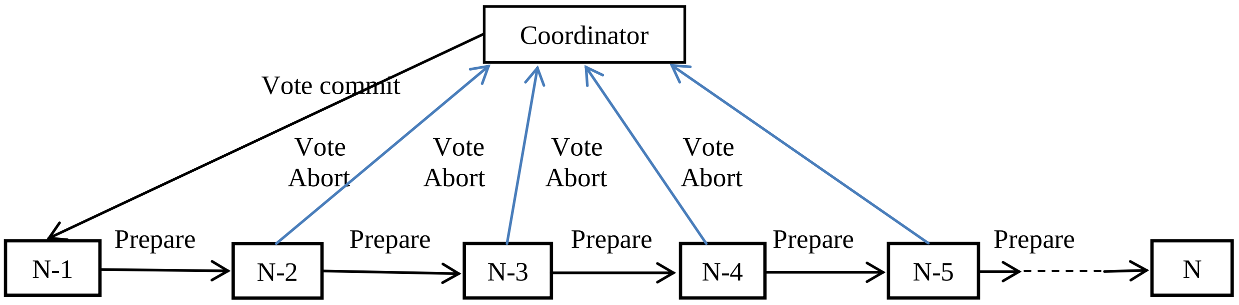

Fig 3: Step One in Linear Two-Phase Commit Protocol

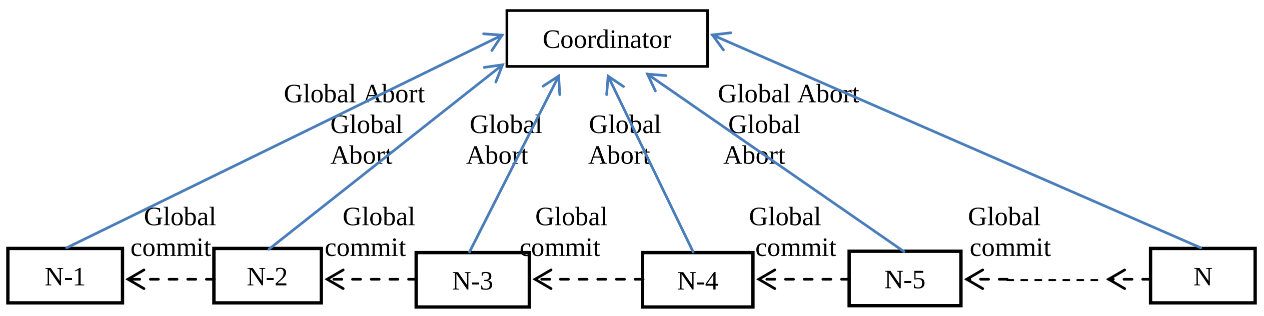

Fig 4: Step two in Linear Two-Phase Commit Protocol

4.3 The Distributed Two-Phase Commit Protocol:

In the distributed 2PC, all nodes communicate with each other. According to this protocol, as you can see on Figure 5, the second phase is not needed as it was present in the previous two 2PC methods. Each node on this method, must have a list of all the participating nodes in order to know that each node has sent in its vote. The distributed 2PC starts when the coordinator sends a PREPARE message to all the participating nodes. When each participant gets the PREPARE message, it sends its vote to all the other participants. This way, each node maintains a complete list of the participants in every transaction. Each participant has to wait and receive the vote from all other participants. When a node receives all the votes from all other participants, it can decide directly on COMMIT or Abort. There is no need to start the second phase, since the coordinator does not have to consolidate all the votes in order to arrive at the final decision.

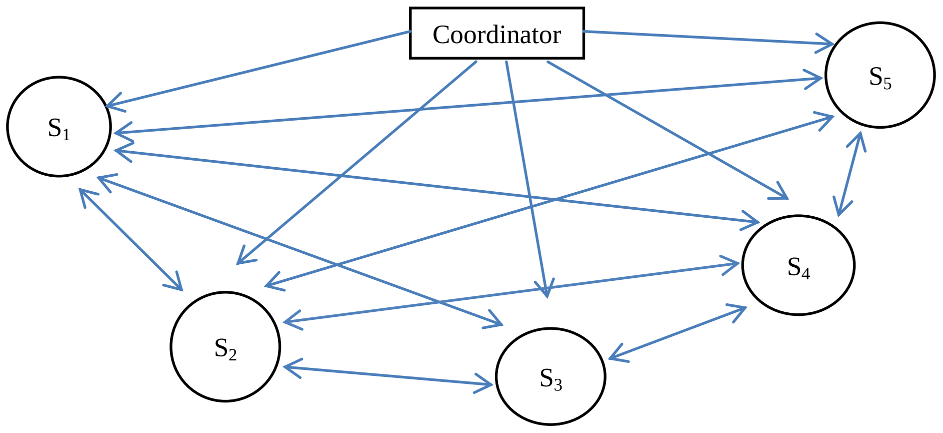

Fig 5: Step two in Distributed Two-Phase Commit Protocol

4.4 Problems with 2PC:

There are two problems with the Two-phase Commit Protocol.

1) Blocking: The Two-Phase Commit Protocol goes to a blocking state by the failure of the coordinator when the participants are in an uncertain state. The participants keep locks on some resources until they receive the next message from the coordinator after its recovery.

2) State Inconsistency: Global state vector in commit protocols works as a container of states for every participating node regarding a single transaction. The global transition state comprises this state vector and outstanding messages in the network.

5. SUGGESTED ALGORITHM FOR DISTRIBUTED TWO-PHASE COMMIT PROTOCOL

From the coordinator:

```
For each participant
    Send PREPARE_MESSAGE
Next
```

```
From participant_N
    Clear ListOfAborts    ;clear list with participants who send an abort
    AbortCount = 0        ;count the number of aborts
    Index = 0             ;index for list with participants
    For each participant
        ;
        If AbortReceived then       ;if abort received from a participant
            Index=Index + 1
            AbortCount = AbortCount + 1
            ListOfAborts(Index)=participant_number
        End if
        If  AbortCount > 0.05*AllParticipants then
            ABORT_TRANSACTION
            Exit sub
        End if
    Next
    ;
    For Index = 0 to AbortCount
        Send PREPARE_MESSAGE        ; send prepare message for second chance to the participants
                                    ; who had send an ABORT
        Wait for vote from participant
        If ABORT_RECEIVED then ABORT_TRANSACTION
    Next Index
End sub
```

In most of the cases, the two-phase commit protocol is blocked because a transaction is blocked due to the coordinator's failure when the participant is in the ready-to-commit state, in my suggested algorithm, I count first the number of Abort votes given from participants. If this Abort votes total number is less than 35 percent of the total participating nodes, then for each one of these participants I suggest to execute the transaction for second time by sending a PREPARE_MESSAGE. Most of these participants will execute the transaction, which means that we will not need to send a PREPARE_MESSAGE to all the participants at a later time to execute this transaction. My conclusion, as it is proven byt the following tables, is that when we apply the suggested algorithm, the total execution time of all transactions involved is much less than the total execution time that was needed when we applied the regular method. This difference will increase considerably when there is a large number of participants.

Example1.

If we suppose that we have 10 participants in a distributed database system and for a transaction requested to execute 3 of the participants give an Abort vote (the last 3 participants according to the order they get the PREPARE_MESSAGE from the coordinator) and if we also suppose that each transaction needs 2 sec to execute, then we can complete the following tables.

Table 1: Execution time applying normal method

Transaction	**Execution time**
Participant-1	2.0 sec
Participant-2	2.0 sec
Participant-3	2.0 sec
Participant-4	2.0 sec
Participant-5	2.0 sec

Participant-6	2.0 sec
Participant-7	2.0 sec
Participant-8	Sends an Abort vote
Participant-9	
Participant-10	
Total spent execution time 14 sec and transaction not completed	

Table 2: Execution time applying my suggested algorithm

Transaction	**Execution time**
Participant-1	2.0 sec
Participant-2	2.0 sec
Participant-3	2.0 sec
Participant-4	2.0 sec
Participant-5	2.0 sec
Participant-6	2.0 sec
Participant-7	2.0 sec
Participant-8	Sends an Abort vote and gets a second chance to complete
Participant-9	Sends an Abort vote and gets a second chance to complete
Participant-10	Sends an Abort vote and gets a second chance to complete
	Total spent execution time 20 sec and transaction completed

This means that in the first case the transaction is not completed and the system lost 14 sec. It will try to execute the same transaction at a later time and the total execution time will be 14 sec + 20 sec = 34 sec.

Using my suggested algorithm, the system will only spend 20 sec to execute the transaction because it gave a second chance for a number (3) of participants who originally had sent an Abort vote.

Example 2.

If we suppose that we have 10 participants in a distributed database system and for a transaction requested to execute 3 of the participants give an Abort vote (the participants 5, 6 and 7 according to the order they get the PREPARE_MESSAGE from the coordinator) and if we also suppose that each transaction needs 2 sec to execute, then we can complete the following tables.

Table 3: Execution time applying normal method

Transaction	**Execution time**
Participant-1	2.0 sec
Participant-2	2.0 sec
Participant-3	2.0 sec
Participant-4	2.0 sec
Participant-5	Sends an Abort vote
Participant-6	
Participant-7	
Participant-8	
Participant-9	
Participant-10	
Total spent execution time 8 sec and transaction not completed	

Table 4: Execution time applying my suggested algorithm

Transaction	**Execution time**
Participant-1	2.0 sec
Participant-2	2.0 sec
Participant-3	2.0 sec
Participant-4	2.0 sec
Participant-5	Sends an Abort vote and gets a second chance to complete
Participant-6	Sends an Abort vote and gets a second chance to complete
Participant-7	Sends an Abort vote and gets a second chance to complete
Participant-8	2.0 sec
Participant-9	2.0 sec
Participant-10	2.0 sec
	Total spent execution time 20 sec and transaction completed

This means that in the first case the transaction is not completed and the system lost 8 sec. It will try to execute the same transaction at a later time and the total execution time will be 8 sec + 20 sec = 28 sec.

Using my suggested algorithm, the system will only spend 26 sec to execute the transaction because it gave a second chance for a number (3) of participants who originally had sent an Abort vote.

Example 3.

If we suppose that we have 10 participants in a distributed database system and for a transaction requested to execute 3 of the participants give an Abort vote (the participants 5, 6 and 7 according to the order they get the PREPARE_MESSAGE from the coordinator) and if we also suppose that each transaction needs 2 sec to execute, then we can complete the following tables.

Table 5: Execution time applying normal method

Transaction	**Execution time**
Participant-1	2.0 sec
Participant-2	Sends an Abort vote and gets a second chance to complete
Participant-3	Sends an Abort vote and gets a second chance to complete
Participant-4	Sends an Abort vote and gets a second chance to complete
Participant-5	2.0 sec
Participant-6	2.0 sec
Participant-7	2.0 sec
Participant-8	2.0 sec
Participant-9	2.0 sec
Participant-10	2.0 sec
Total spent execution time 2.0 sec and transaction not completed	

Table 6: Execution time applying my suggested algorithm

Transaction	**Execution time**
Participant-1	2.0 sec
Participant-2	Sends an Abort vote and gets a second chance to complete
Participant-3	Sends an Abort vote and gets a second chance to complete
Participant-4	Sends an Abort vote and gets a second chance to complete
Participant-5	2.0 sec
Participant-6	2.0 sec
Participant-7	2.0 sec
Participant-8	2.0 sec
Participant-9	2.0 sec
Participant-10	2.0 sec
	Total spent execution time 20 sec and transaction completed

This means that in the first case the transaction is not completed and the system lost 8 sec. It will try to execute the same transaction at a later time and the total execution time will be 2 sec + 20 sec = 22 sec.

Using my suggested algorithm, the system will only spend 20 sec to execute the transaction because it gave a second chance for a number (3) of participants who originally had sent an Abort vote.

6. CONCLUSIONS

Distributed database systems widely use two-phase commit protocol for their transactions over the network which means that it is essential to verify their correctness and speed. Applying my suggested distributed two-phase commit protocol, we achieve a significant gain in time executing transactions when an abort vote happens from some of the participants.

This is proved by the three examples sown above where I examine 3 different cases of transactions that are aborted and the results are shown in tables from 1 through 6.

REFERENCES

[1] Buretta, Marie. (1997) Data Replication, New York: John Wiley & Sons

[2] Burleson, Donald K. (1994) Managing Distributed Databases. New York: John Wiley & Sons

[3] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," IEEE Trans. Software Eng., vol. 9, no. 3, pp. 219–228, 1983.

[4] D. Skeen, "Nonblocking commit protocols," in SIGMOD Conference, Y. E. Lien, Ed. ACM Press, 1981, pp. 133–142.

[5] "A quorum-based commit protocol," in Berkeley Workshop, 1982, pp. 69–80.

[6] T. Härder and A. Reuter, "Principles of transaction-oriented database recovery," ACM Comput. Surv., vol. 15, no. 4, pp. 287–317, 1983.

[7] Ozsu, Tamer M., and Valduriez, Patrick [1991], Principles of Distributed Database Systems, Prentice Hall.

[8] Abraham Silberschartz, Henry F. Korth (2006) Database SystemConcepts, Mc Graw Hill International edition.

[9] T. Härder and A. Reuter, "Principles of transaction-oriented database recovery," ACM Comput. Surv., vol. 15, no. 4, pp. 287–317, 1983.

[10] Ramakrishnan, Gehrke Database Management Systems, Mc Graw Hill International edition (2003)

[11] Bhargava Bharat: Concurrency Control in Database systems: IEEE Transactions on knowledge and data engineering, VOL.11, No1,1999 (IEEE)

[12] Ali R. Abdou, Hany M. Harb: Two phase locking concurrency control in distributed databases with N-Tier architecture; IEEE 2004.

[13] Bhargava Bharat: Concurrency Control in Database systems: IEEE Transactions on knowledge and data engineering, VOL.11, No1,1999 (IEEE)

[14] "A quorum-based commit protocol," in Berkeley Workshop, 1982, pp. 69–80.

[15] A. Bestavros, "Multi-version Speculative Concurrency Control with Delayed Commit", P m . of Zntl. Conf. on Computers and their Applications, March 1994.

[16] E. Cooper, "Analysis of Distributed Commit Protocols", Proc. of ACM Sigmod Conf., June 1982.

[17] R. Gupta, J. Haritsa, K. Ramamritham and S. Seshadri, "Commit Processing in Distributed RTDBS", TR-96-01, DSL/SERC, Indian Institute of Science

[18] J. Haritsa, M. Carey, and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems", Real-Time Systems Journal, 4 (3), 1992.

[19] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," IEEE Trans. Software Eng., vol. 9, no. 3, pp. 219–228, 1983.

[20] 0. Ulusoy and G. Belford, "Real-Time Lock Based Concurrency Control in a Distributed Database System", Proc. Of 12th Zntl. Conf. on Distributed Computing Systems, 1992.

[21] C. Mohan, B. Lindsay and R. Obermarck, "Transaction Management in the R* Distributed Database Management System", ACM TODS, 11(4), 1986.

[22] D. Skeen, "Nonblocking commit protocols," in SIGMOD Conference, Y. E. Lien, Ed. ACM Press, 1981, pp. 133–142.

[23] Ulusoy, O. (1994). "Processing real-time transactions in a replicated database system. In Distributed and Parallel Databases", volume 2, pages 405–436.

[24] Son, S. (1987). "Using replication for high performance database support in distributed real-time systems". In the 8th IEEE Real-Time Systems Symposium, pages 79–86.

[25] Y. Yoon, "Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems", Ph. D. Thesis, Korea Adv. Inst. of Science and Technology, May 1994.