www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 4 Issue 5 May 2015, Page No. 12188-12193

Extended Two Phase Commit Protocal In Real Time Distributed Database System

Sanjeev Kumar Singh Kushwaha
Babu Banarasi Das University, BBD City, Faizabad road, Lucknow, U.P
Sanjeevkushwaha30@gmail.com

ABSTRACT: The two phase commit (2pc) protocol is widely used for commit processing in distributed data base system (DDBSs). The blocking problem in 2pc reduces the availability of the system as the blocked transaction keeps all the resources until receive the final command from the coordinator after its recovery. To remove the blocking in 2pc, three phase commit (3pc) protocol was proposed. Although 3pc protocol eliminates the blocking problem, it involves an extra round of message transmission, which degrades system performance in DDBSs. Both 2pc and 3pc having problem which degrades system efficiency .In order to remove this problem in 2pc and 3pc ,E2PCP protocol was introduce to enhance system performances as compare to 2pc and 3pc.
To reduce blocking, we propose an extended two phase commit protocol (E2PCP) by attaching multiple participant sites to the coordinator sites work as a backup sites or as substitute sites for coordinator sites. In this protocol, after receiving responses from all participant sites in the first phase, the coordinator communicates the final decision to the backup sites in the back phase. Afterward, it send final decision to the participants. When blocking occur due to failure of coordinator site, the participant site can terminate the transaction by consulting backup sites of the coordinator. In this way E2PCP protocol achieving non-blocking in most of coordinator sites failures.

Keyword: distributed database, commit protocol, 2phase commit, 3 phase commit, extended two phase commit protocol (E2PCP), distributed algorithm

INTRODUCTION: In distributed database system transaction is of important element in distributed system like airline reservation systems, banking applications, credit-card systems, and stock-market transactions, widely use these protocols for their transactions over the network. So, undoubtedly, it is essential to improve transaction processes and to verify their correctness. So that process get completed in given time period and can increases system performance. Basically transaction are associated with deadlines. Meeting deadlines is one of the important objectives.
Real time database system operating on distributed data have to contend with complexities of transaction ACID semantics in distributed data. Every transaction process system must ensure this ACID property for successful transaction process. ACID property stand for:

Atomicity guarantees that many operations are bundled together and appear as one contiguous unit of work, operating under an all-or-nothing paradigm—either all of the data updates are executed, or nothing happens if an error occurs at any time. In other words, in the event of failure in any part of the transaction, all data will remain in its former state as if the transaction was never attempted. In transactional terminology, this is referred to as *rolling back* the transaction.

Consistency guarantees that a transaction will leave the system in a consistent state after the transaction is completed. The meaning of consistency varies depending on the logic of the system; it is somewhat up to the application developer to enforce the specific rules governing the consistent state.
Within a transaction, it is possible for some pieces to be in an inconsistent state. However, once the transaction is completed—either successfully or unsuccessfully—the system must return to a consistent state. An example most of us can relate to is a software application installer. Installers write and update files on your hard drive. If you should turn off your computer in the middle of an installation you may be unable to continue the installation or uninstall the program without some manual manipulation of your file system and/or system registry. The installation of the software was left in an *inconsistent* state. Atomicity helps enforce that the system always appear in a consistent state.

Isolation protects concurrently executing transactions from seeing each other's incomplete results. Isolation allows multiple transactions to read or modify data without knowing about each other because each transaction is isolated from the others. This is achieved by using low level synchronization protocols (locking) on the underlying data. There are several levels of isolation available, each with benefits and drawbacks. For example, at the lowest level of isolation, as the data is being changed in the transaction, other users of the data will be exposed to the changes. Thus, if the transaction is rolled back, the other users of the data may see data that will not be accurate a few moments later after the roll back occurs. At higher levels of isolation, other users of the data will not be able to read the data until the transaction is successfully completed or is rolled back.

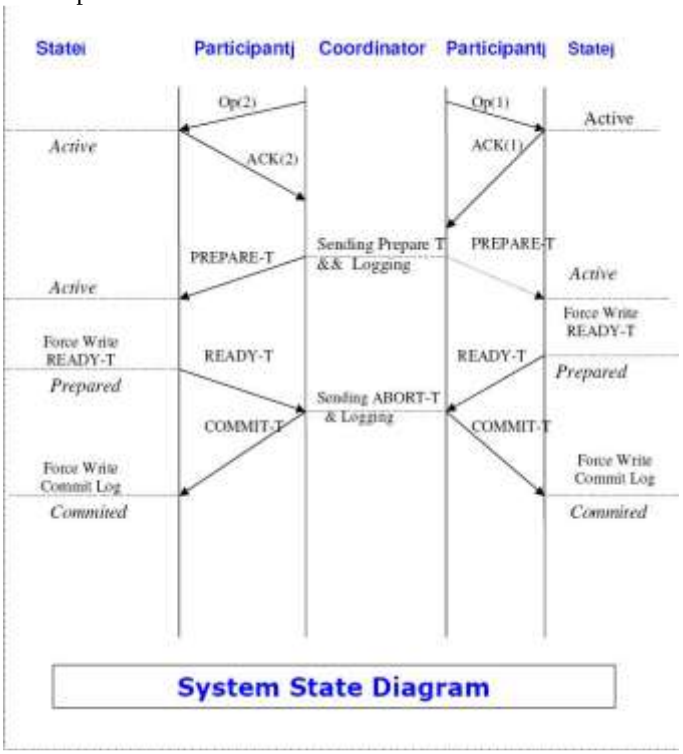
Durability guarantees that updates to managed resources survive failures. Failures include machine crashes, network crashes, hard disk crashes, and power failures. Recoverable resources keep a transactional log so that the permanent data can be reconstructed by reapplying the steps in the log.
The lifetime of a transaction is divided into two stages: the execution stage and another one is the commitment stage. In execution stage, the operation of transaction are processed a different sites of the system, while in the commitment stage, a commit protocol is executed to ensure failure atomicity. The transactions in the stage are called *executing transactions* and the transactions in the commitment stage are called *committing transactions*. There are several important factors contributing to the difficulty in meeting the transaction deadlines in a DRTDBS. In two phase commit protocol blocking transaction may seriously affect the performance of a DRTDBS, especially when failure occur during the commitment phase. Due to the delay caused by the failures, the blocked transaction may have a high probability of missing their deadlines. In the 2PC, the process of a transactions at different sites are divided into two groups. One of the processes is the coordinator and the other are the participants. The following factors can causes a long delay in the execution of 2PC:

1-Unpredictable communication delays. Since the 2pc requires at least two rounds of message communications between the coordinator and the participants, its performance is highly dependent on the performance of the underlying network, the communication delays are still unpredictable due to loss of messages or failures of communication links.

2-Failure of coordinator and participants. Different kinds of failures may occur in the coordinator and in the participants during the execution of the commit protocol. Although the 2pc is resilient to these failures, the resolution methods are usually based on time –out. However, it is not easy to determine a suitable time-out period for resolving the failures. A well-chosen time-out interval is important to the performance of a real-time system. Otherwise, an executing transaction, which is blocked by a committing transaction, can be blocked for a very long time before the system detects the failure. The above factors not only affect the performance of the transactions in the execution stage, but also they have a serious impact on the performance of committing transactions because these transaction are close to their completions and some of their participants might be committing.

I. TWO PHASE COMMIT PROTOCOL (2PC)

The 2-phase commit (2PC) protocol is a distributed algorithm to ensure the consistent termination of a transaction in a distributed environment. Thus, via 2PC a unanimous decision is reached and enforced among multiple participating servers whether to commit or abort a given transaction, thereby guaranteeing atomicity. The protocol proceeds in two phases, namely the prepare (or voting) and the commit (or decision) phase, which explains the protocol's name. The protocol is executed by a coordinator process, while the participating servers are called participants. When the transaction's initiator issues a request to commit the transaction, the coordinator starts the first phase of the 2PC protocol by querying—via prepare messages—all participants whether to abort or to commit the transaction.
If all participants vote to commit then in the second phase the coordinator informs all participants to commit their share of the transaction by sending a commit message. Otherwise, the coordinator instructs all participants to abort their share of the transaction by sending an abort message. Appropriate log entries are written by coordinator as well as participants to enable restart procedures in case of failures.



Problems with 2PC

There are two problems with the above-described Two-Phase Commit Protocol.

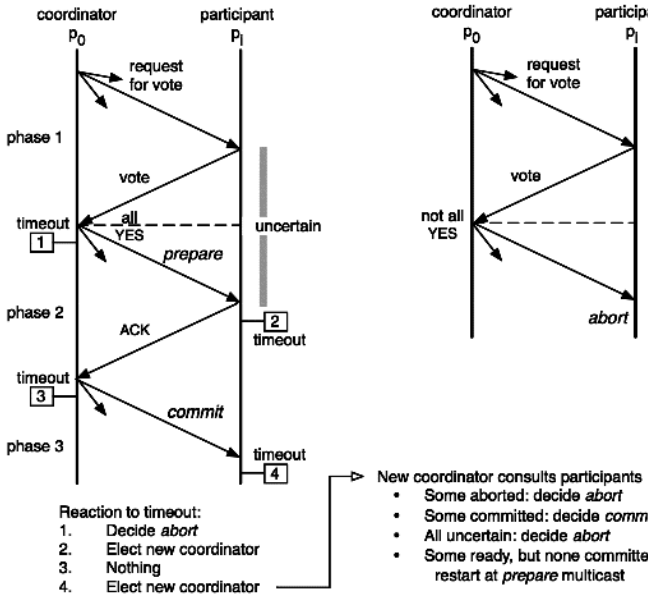
1) *Blocking:* The Two-Phase Commit Protocol goes to a blocking state by the failure of the coordinator when the participants are in uncertain state. The participants keep locks on resources until they receive the next message from the coordinator after its recovery.

2) *State Inconsistency:* Global state vector in commit protocols works as a container of states for every participating node regarding a single transaction. When its global state vector contains both the commit and abort states. This inconsistency can be observed using a state vector, particularly when the participant is at its pre-commit state (p2) and fails. The coordinator shows the committed state after sending *commit* message but for the failed participant the protocol is declared non resilient for assigning new state. It involves a great deal of message complexity.
• Greater communication overheads as compared to simple optimistic protocols.
• Blocking of site nodes in case of failure of coordinator.
• Multiple forced writes of log, which increase latency.
• Its performance is again a tradeoff, especially for short lived transactions, like Internet applications.

II. THE THREE-PHASE COMMIT PROTOCOL (3PC)

Three-Phase Commit Protocol (3PC) is a non-blocking Protocol, contrary to the 2PC. Here a new state called "precommit" is introduced for the coordinator in [2]. The coordinator gets to this "pre-commit" state only if all other participants have voted to commit, i.e., *yes*. In case this state is

not reached, the participant will abort and release the blocked resources after a specific time. When the coordinator gets the "pre-commit" state then there is only one option to abort the transaction and that is a timeout, which corresponds to a failure of a participant, otherwise the transaction gets completed with an acknowledgement from the participants. It is also possible that the coordinator fails at this state, even then it will proceed for global commit as shown in Figure 3PC with failure and timeout transitions [2]



B. Problems with 3PC

Three-Phase Commit Protocol is problematic only when there are multiple sites failures (proved in section VI-B).

For example, let's consider a case where the coordinator is in "pre-commit" state and fails just after sending a *commit* message and the slave also fails just before or after receiving this message as shown in Figure 5. So by its failure, the slave moves to the aborted state but according to the protocol specifications given in [3], the coordinator goes to the committed state, either it fails or receives acknowledgement. Hence, the coordinator moves to the committed state without receiving acknowledgement and the failed slave moves to the aborted state without sending the acknowledgement. In this way, coordinator and participant show different final states due to their failures.

III. DISTRIBUTED TRANSACTION:

Transaction may access data at several sites.

• **Each site has a local transaction manager responsible for**:

– Maintaining a log for recovery purposes

– Participating in coordinating the concurrent execution of the transactions executing at that site.

• **Each site has a transaction coordinator, which is responsible for:**

– Starting the execution of transactions that originate at the site.

–Distributing sub transactions at appropriate sites for execution.

–Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.

Distributed transaction processing systems are designed to facilitate transactions that span heterogeneous, transaction-aware resource managers in a distributed environment. The execution of a distributed transaction requires coordination between a global transaction management system and all the local resource managers of all the involved systems. The resource manager and *transaction processing monitor (*or *TPM* as used herein) are the two primary elements of any distributed transactional system. The TPM is responsible for managing distributed transactions by coordinating with different resource managers to access data from several different systems. Since multiple application components and resources participate in a transaction, it is necessary for the TPM to establish and maintain the state of the transaction as it occurs. Resource managers inform the TPM of their participation in a transaction by means of a process called *resource enlistment*. The TPM keeps track of all the resources participating in a transaction and uses this information to coordinate transactional work. The TPM has to monitor the execution of the transaction and determine whether to commit or roll back the changes made to ensure atomicity of the transaction.

Elements of an Extended Two-Phase Commit Protocol in Real Time Distributed Database System Definitions for the various elements of an E2PCP system are provided below:

• **Application** Software can be defined as a program or group of programs designed for end users. Software can be divided into two general classes: *systems software* and *applications software*. Systems software consists of low-level programs that interact with the computer at a very basic level. This includes operating systems, compilers, and utilities for managing computer resources. In contrast, application software (also called *end-user programs*) includes database programs, word processors, and spreadsheets. Figuratively speaking, application software sits on top of systems software because it is unable to run without the operating system and system utilities.

• **Resource Manager (RM)** The resource manager is usually a database management system, such as Oracle, DB2, or SQL Server. A resource manager is responsible for maintaining and recovering its own resources. From the perspective of the application, the resource manager is a single attachment to the resource (e.g., a database). Note that resource managers are not limited to databases. Any software program that manages persistent data is a resource manager.

• **Transaction Manager (TM)** The transaction manager coordinates the actions of the resource managers that are located on the same node (local resource managers) as the transaction manager. (A transaction manager may also act as the coordinator under specific circumstances.)

The transaction manager should implement this interface so that the code for the commit protocol can be plugged in the Simputer DB without any modification. The transaction manager that we have assumed is capable of handling multiple transactions. We maintain a Link List of Transaction states modified atomically. This is ensured using semaphores. For each transaction we assign a transaction Id.

The functions that the interface contains are:

1. Void Start Transaction (Transaction ID) This method is used to initialize the transaction. While implementing the actual transaction manager you may need to change the prototype to initialize the transaction.

2. Void Acquire Resources (Transaction ID)
We have provided this dummy function so that the functions to be called for initializing the transaction, to acquire locks on the resources etc. can be called within this function. And this function can be called from the Start Transaction Method as we currently do.

3. Void Release Resources (Transaction ID)
We have provided this function as an interface to all the steps required to be executed while ending the transaction, like releasing the locks, freeing the memory, deleting or force
Writing the remaining logs etc. Currently we delete the remaining logs.
4. In getTransactionId ( ) the implementation of this method can be modified to meet the requirement of the transaction Manager. It returns the integer value for the transaction id of the new transaction.

5. int tiggerDefferedConstraints (Transaction ID This function is invoked during the precommit phase. It can be used for invoking the code for checking the deferred constraints.

6. int Create Log (Transaction Id ,log Type , message , coordinator ID)This function has been provided to be invoked from the functions for commit protocol.

This has been done keeping in mind that the structure of log records may change with the actual implementation of the log manager. To avoid any modification in the code of the commit protocol, this method acts as an interface for creating and inserting the log records. With the change in the log structure only this function has to be changed.

• **Transaction Coordinator (TC)** The transaction coordinator is the transaction manager on the node where the application started the transaction. The coordinator orchestrates the distributed transaction by communicating with transaction managers on other nodes (remote transaction managers) and with resource managers on the same node (local resource managers).
• **Transaction Processing Monitor (TPM)** The transaction processing monitor consists of the transaction coordinator and all the transaction managers composing the distributed E2PCP system.

IV. LOG MANAGER:

The log manager has been implemented keeping in mind the requirement of the actual log manager. The code for the log manager can be reused as far as the insert, flush and delete are the requirements. The logs are maintained as a link list in the memory. The pointer to the last record inserted is maintained for the fast insertion of the log record. The records contain a pointer to the next log record for the same transaction. Thus the log records for all the transactions are in the same list, still tracing through the log records for a particular transaction is optimal and Direct. It do not require traversing through the log records of other transaction. This makes flushing optimal. We have implemented three functions for log manager.
1**. Void insert Log (Log Record)** This function takes in the log Record and inserts it into the log. Even if the structure of the log Record changes you need not change the implementation of the function.

2**. Void flush Log (Transaction ID)** This function flushes all the log records for a given transaction Id on to the stable storage and frees the memory.
3. **Void delete Log (Transaction ID)** This function deletes all the log records in the memory for a transaction with the given transaction id.
3.6 Languages Tools and Libraries

V. ALGORITHM:

The protocol involves all the local sites at which the transaction executed. Let *T* be a transaction initiated at site *Sj* and let the transaction coordinator at *Sj* be *Cj*.

Phase 1:

- *Cj* adds <*query to commit*> record to the log.
- *Cj* sends <*query to commit*> message to all sites.
- When a site receives a <*query to commit*> message, the Transaction manager determines if it can commit the transaction.
- If no: Add <*no T*> record to the log and respond to *Cj* with <*abort T*>.
- If yes: Add <*ready T*> record to the log, force *all*

Log records for *T* onto stable storage and transaction manager sends <*ready T*> message to *Cj*.

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

Phase 2: The following are the actions performed during this phase:

- The coordinator adds a decision record <*abort T*>or <*commit T*> to its log and forces the record onto stable storage.
- Once that record reaches stable storage it is irrevocable (even if failures occur).
- *Cj* communicates the final decision to the backup sites in the backup phase
- The coordinator sends a message to each participant informing it of the decision (*commit* or *abort*).
- If *Cj* failed
  {
  blocking occur
  }
- Cohorts/participant sites <terminate T> consulting multiple backup site of *Cj*

Site failure in E2PC is handled in the following manner:

- If the log contains a <*commit T*> record, the site executes *redo (T)*.
- If the log contains an <*abort T*> record, the site executes undo (T).

- If the log contains a <*ready T*> record, consult *Cj*. If *Cj* is down, site sends *query-status T* message to the other sites.

If the log contains no control records concerning *T*, the site executes *undo (T)*.

VI. PROBLEM EVALUATION AND ANALYSIS:

Simulation was done for both the main memory resident and disk resident databases at communication delay of 0ms and 100ms .we compare E2PCP with 2pc and 3pc in this experiment. Following figure show the miss percent behavior under normal and heavy load conditions with/without communication delay and figure deal with main memory based database system while rest of the figure deal with disk resident database system.

Figure 1: Miss% with (RC+DC) at communication delay 0ms normal and heavy load.

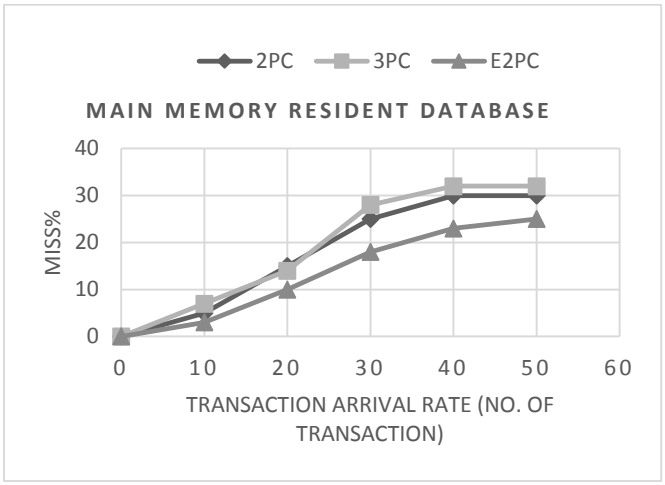


Figure 2: Miss% with (RC+DC) at communication delay 100ms normal and heavy load.

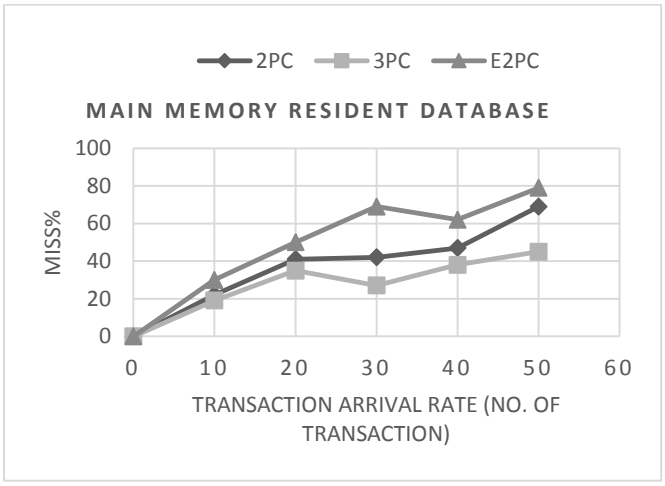


Figure 3: Miss% with (RC+DC) at communication delay 0ms normal load.

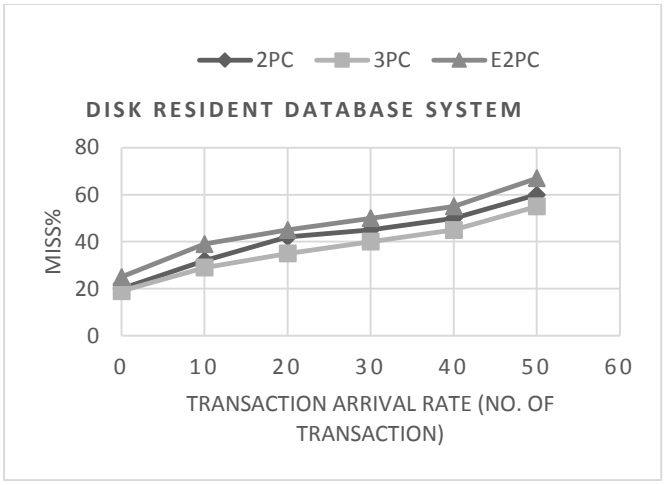


Figure 4: Miss% with (RC+DC) at communication delay 0ms and heavy load.

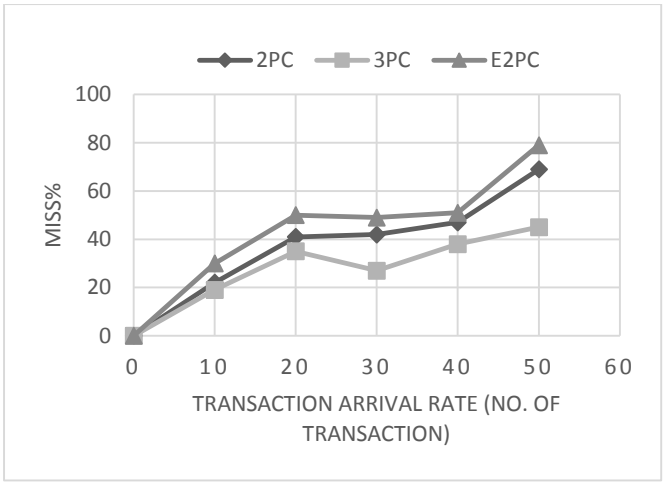
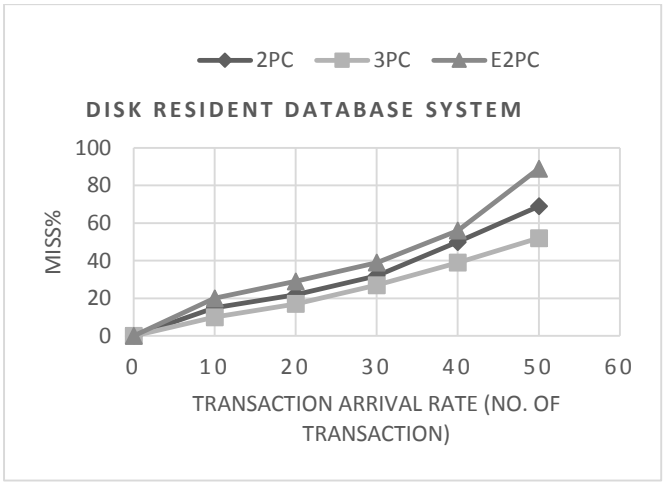


Figure 5: Miss% with (RC+DC) at communication delay 100ms normal and heavy load.



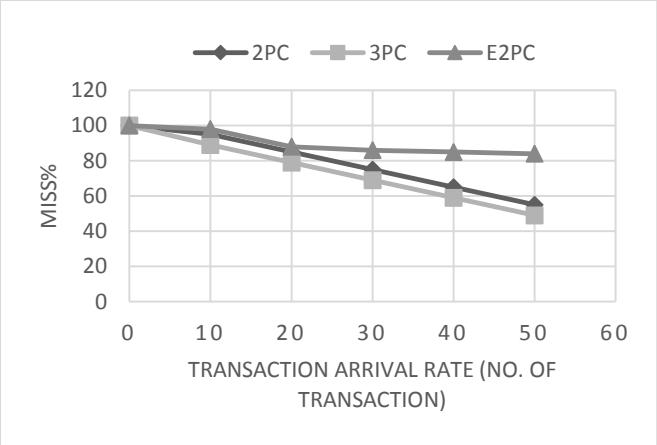


Figure 6: success ratio

VII. CONCLUSION:

In this paper we have proposed new protocol E2PCP.This protocol basically solve the problem of blocking in 2 phase commit and also solve the problem of 3pc in which extra round of transaction get increases which reduces system efficiency and performance. This protocol eliminate both the problem of 2pc and 3pc which reduce transaction failure and solve the problem of miss transaction in distributed transaction system. This protocol increases efficiency of transaction system .The performance of E2PCP is compared with other protocol for both main memory resident and disk resident databases without communication delay.

VIII. REFERENCES:

[1] Udai Shankar, Nikhil, Shalabh, Praveen, Praphul Srivastava, "ACTIVE-Areal Time Commit Protocol".

[2] "The PROMT -Real Time Commit Protocol", Jayant R. Haritsa, Krithi Ramamritham, Ramesh Gupta

[3] "Active Real Time Protocol", Udai Shanker, Nikhil Agarwal, Shalabh Kumar Tiwari, Praveen Srivastava

[4] Udai Shanker, Manoj Misra, Anil K. Scare and Rahul Shisondia "Dependency Sensitive Shadow

[5] SWIFT", 10thInternational Database Engineering and Application Symposium IEEE 2006.

[6] Poonam Singh, Parul Yadav, Sanchit Lohia "An Extended 3 phase commit protocol for concurrency control in distributed system" IJCA 2011 volume 21-No.10.

[7] S. Agrawal, Udai Shanker, Abhay N. Singh, A. Anand, "©2010 International Journal of Computer Applications (0975 – 8887) Volume 1 – No. 3.

[8] J. R. Haritsa, K. Ramamritham, and R. Gupta, "The PROMPT real time commit protocol," IEEE Transaction on parallel and distributed systems, 11(2), 2000, pp. 160-181

Author Profile:

Sanjeev Kumar Singh Kushwaha received the BTech degrees in computer science and engineering from Uttar Pradesh Technical University in 2011 and pursing MTech(final year) from Babu Banarasi Das University..