

Issues in Concurrency Control for Different Databases Concurrency Control Methods for Different Databases a Survey of Concurrency Control Protocols

Gitanjali Mishra* Jyotirmaya Mishra

Assistant Professor, Computer Science and Engineering,
Gandhi Institute of Engineering & Technology, Gunupur, Odisha, INDIA.

Abstract: This paper reviews the coverage of concurrency control in different Databases. Database becomes more popular, the need for improvement in database management systems becomes even more important to maintain reliability. The main challenges are identified as-: (1)Preserving the ACID property atomicity, consistency, isolation and durability property when concurrent transactions perform read and write operation; (2) provides recovery method when distributed data is fail; (3)whatever method that is chosen they must provide feasible solutions with respect to performance.

Keywords:Centralized Database system, Distributed Database System, object-oriented database System, Distributed Object-Oriented Database System , Mobile Database System, Real-Time Database System ,Multilevel Secure Databases, Replicated Real Time Databases, Concurrency control, Transaction, Locking protocol

I. INTRODUCTION

In today's world of universal dependence on information systems, with the rising need for secure, reliable and accessible information in today's business environment, the need for databases and client/ server applications is also increasing. This paper reviews the concurrency control in dissimilar databases. Database becomes more trendy and to supervise different types of database management systems are required. Many transactions are accessing the databases concurrently. The main challenges are identified for the transactions are -: (1)Preserving the ACID property atomicity, consistency, isolation and durability property when concurrent transactions perform read and write operation; (2) provides recovery method when data is failed; (3)whatever method that is chosen they must provide feasible solutions with respect to performance. One most important mechanism to control the concurrent transactions is concurrent control mechanism. Here we are focusing different types of concurrency control mechanism for different databases. In this paper we address lock concept in different databases transactions.

II. CONCURRENCYCONTROL IN CENTRALIZED DATABASESYSTEM

The basic idea of locking is that whenever a transaction accesses a data item, it locks it, and that a transaction which wants to lock a data item which is already locked by another transaction must wait until the other transaction has released the lock (unlock).

Let us see some important terminologies related to this concept:

- **Lock Mode:** Transaction locks the data item in the following modes:
 - **Shared Mode:** Here the transaction wants only to read the data item.
 - **Exclusive Mode:** Here the transaction wants edit the data item.
- **The Well-formed Transactions:**The transactions are always well-formed if it always locks a data item in shared mode before reading it, and it always locks a data item in exclusive mode before writing it
- **Compatibility Rules existing between Lock Modes:**
 - A transaction can lock a data item in shared mode if it is not locked at all or it is locked in shared mode by another transaction
 - A transaction can lock a data item in exclusive mode only if it is not locked at all.
- **Conflicts:** Two transactions are in conflict if they want to want to lock the same data item with two compatible modes; two types of conflicts: Read-Write conflict and Write-Write conflict.
- **Granularity of Locking:** This term relates to the size of objects that are locked with a single lock operation. In general, it is possible to lock at the *record level* (i.e to lock individual tuples)or at the *File level* (to lock at the fragment level).
- **Concurrent transactions are successful if the following rules are followed:**
 - mTransactions are well-formed
 - Compatibility rules are observed

- Each transaction does not request new locks after it has released a lock.

A sophisticated locking mechanism known as *2-Phase locking* which includes the above said principles is normally used. According to this, there are two separate phases:

- *Growing phase*: Each transactions there is a first phase during which new locks are acquired
- *Shrinking Phase*: A second phase during which locks are only released.

We will simply assume that all transactions are performed according to the following scheme:

(Begin Application)
Begin transaction
Acquire locks before reading or

writing

Commit
Release locks
(End application)

In this way the transactions are well formed, 2-Phase locked and isolated.

Deadlock: A deadlock between two transactions arises if each transaction has locked a data item and is waiting to lock a different data item which has already been locked by the other transaction in the conflicting mode. Both transactions will wait forever in this situation, and system intervention is required to unblock the situation. The system must first find out the deadlock situation and force one transaction to release its locks, so that the other one can proceed. i.e one transaction is aborted. This method is called as **Deadlock detection**.

III. CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEM

In distributed database systems, database is typically used by many users. These systems usually allow multiple transactions to run concurrently i.e. at the same time. Concurrency control is the activity of coordinating concurrent accesses to a database in a multiuser database management system (DBMS). Concurrency control permits users to access a database in a multi-programmed fashion while preserving the illusion that each user is executing alone on a dedicated system. The main technical difficulty in attaining this goal is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another. When the transactions are updating data concurrently, it may lead to several problems with the consistency of the data.

Distributed Transaction:

A distributed transaction is a transaction that runs in multiple processes. 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 are the two primary elements of any distributed transactional system. Distributed transactions, like local transactions, must observe the ACID properties. However, maintenance of these properties is very complicated for distributed transactions because a failure can occur in any process. If such a failure occurs, each process must undo any work that has already been done on behalf of the transaction. A distributed transaction processing system maintains the ACID properties in distributed transactions by using two features:

- Recoverable processes: Recoverable processes log their actions and therefore can restore earlier states if a failure occurs.
- A commit protocol: A commit protocol allows multiple processes to coordinate the committing or aborting of a transaction. The most common commit protocol is the two-phase commit protocol.

Distributed Two-Phase Locking (2PL):

In order to ensure serializability of parallel executed transactions states different methods of concurrency control. One of these methods is locking method. There are different forms of locking method. Two phase locking protocol is one of the basic concurrency control protocols in distributed database systems which will ensure serializability. The main approach of this protocol is "read any, write all". Transactions set read locks on items that they read, and they convert their read locks to write locks on items that need to be updated. To read an item, it suffices to set a read lock on any copy of the item, so the local copy is locked; to update an item, write locks are required on all copies. Write locks are obtained as the transaction executes, with the transaction blocking on a write request until all of the copies of the item to be updated have been successfully locked. All locks are held until the transaction has successfully committed or aborted [3].

The 2PL Protocol oversees locks by determining when transactions can acquire and release locks. The 2PL protocol forces each transaction to make a lock or unlock request in two steps:

- *Growing Phase*: A transaction may obtain locks but may not release any locks.
- *Shrinking Phase*: A transaction may release locks but not obtain any new lock.

The transaction first enters into the Growing Phase, makes requests for required locks, then gets into the Shrinking phase where it releases all locks and cannot make any more requests. Transactions in 2PL Protocol should get all needed

locks before getting into the unlock phase. While the 2PL protocol guarantees serializability, it does not ensure that deadlocks do not happen. So deadlock is a possibility in this algorithm, Local deadlocks are checked for any time a transaction blocks, and are resolved when necessary by restarting the transaction with the most recent initial startup time among those involved in the deadlock cycle. Global deadlock detection is handled by a “Snoop” process, which periodically requests waits-for information from all sites and then checks for and resolves any global deadlocks.

Wound-Wait (WW):

The second algorithm is the distributed wound-wait locking algorithm. It follows the same approach as the 2PL protocol. The difference lies in the fact that it differs from 2PL in its handling of the deadlock problem: unlike 2PL protocol, rather than maintaining waits-for information and then checking for local and global deadlocks, deadlocks are prevented via the use of timestamps in this algorithm. Each transaction is numbered according to its initial startup time, and younger transactions are prevented from making older ones wait. If an older transaction requests a lock, and if the request would lead to the older transaction waiting for a younger transaction, the younger transaction is “wounded” – it is restarted unless it is already in the second phase of its commit protocol. Younger transactions can wait for older transactions so that the possibility of deadlocks is eliminated [3].

Example:- Wound-Wait algorithm

	T1 is allowed to
$t(T1) > t(T2) \longrightarrow$	Wait
$t(T1) < t(T2) \longrightarrow$	Abort and rolled back

$t(T1) > t(T2)$:- If requesting transaction $[t(T1)]$ is younger than the transaction $[t(T2)]$ that has holds lock on requested data item then requesting transaction $[t(T1)]$ has to wait.
 $t(T1) < t(T2)$:- If requesting transaction $[t(T1)]$ is older than the transaction $[t(T2)]$ that has holds lock on requested data item then requesting transaction $[t(T1)]$ has to abort or rollback.

IV. CONCURRENCY CONTROL IN OBJECT-ORIENTED DATABASESYSTEM

An object-oriented database management system is defined as a collection of classes and instances of these classes. A class contains the definitions of the variables that will take values in the instances of this class together with the methods used to access these variables. A method execution is considered to be a partial order of sub method calls. It is

assumed that the database environment allows for extensibility, permitting users to dynamically modify the class definitions. As such, we can regard class definitions as objects accessible by the users of the database. Both the class objects and the instance objects (objects derived by instantiating a class object) will be referred to as objects thus providing uniform treatment for all objects in the database. The objects are assumed to be autonomous entities, internally concurrent, with full control over the methods they are running at any time. They are organized in a hierarchy. Relationships among methods are described in terms of standard tree terminology (recursive method calls are not allowed) with a method $m2$ being a child of a method $m1$ if $m1$ invokes $m2$. The notions of parent and ancestor are defined in the same manner. In this environment the methods of one object can invoke only methods of objects that are lower the hierarchy and every object inherits the methods of all its ancestors.

As mentioned before, in a OODBMS a transaction (a user program) consists of a series of method invocations on different objects, which in turn can invoke other methods on different objects, this leading to a tree structure of method calls. As in traditional database systems a transaction is represented as a partially ordered set of method calls which are related among them by conflict, commutatively and concurrency conditions specified by the OODBMS designer or by the users who extend the database with new objects.

The Basic Protocol

The protocol presented in this paper is an extension of the protocol introduced by Agrawal et al [47].

1. A method t' can execute an atomic operation t on an object o if it can acquire a lock on o .
2. A method execution cannot commit until all its children have terminated. When a method terminates:

- If it is not the top-level, its locks are inherited by its parent.
- If it is not top-level and it aborts, its locks are discarded.
- If it is top-level, its locks are discarded.

3. A lock on an atomic operation o is granted to a method if and only if :

- The current state of the object permits the execution of the requesting method.
- If there exist non-ancestors methods holding inherited locks on o , there are some ancestors of these methods and the requesting method that commute.
- Granting locks and scheduling for execution in the same time all other concurrently runnable methods preserves the partial order devised by the central transaction manager.

The novelty of this protocol consists of forcing parent methods to inherit the locks of terminated children methods, allowing conflicting operations to share locks if they have commuting ancestors and permitting objects to execute more than one method call at a time, according to the specifications designed by the users and the system designer.

V. CONCURRENCY CONTROL IN DISTRIBUTED OBJECT-ORIENTED DATABASE SYSTEM

One scheduler module is available in this system, that is a *black box*, it communicates with the other modules only via a well defined interface. In the distributed version of the simulator, two-phase locking and timestamp ordering schedulers are implemented. The scheduler receives an operation from the transaction manager, and processes it according to its scheduling technique. When the scheduler decides that an operation can be sent to the data manager, the data manager is called. When the operation has completed, the scheduler will be notified by a call from the data manager. The following distributed schedulers are implemented:

- **Strict Two-Phase Locking Scheduler:**
An nice feature with non-replicated distributed databases is that each local scheduler can schedule the data accesses as if it was a centralized scheduler. But of course there are also some problems that are more difficult to solve in the distributed context than for a centralized scheduler. For the 2PL scheduler the main problem which has to be solved is deadlock.
- **Strict Timestamp Ordering Scheduler:**
We have implemented a strict TO scheduler. Although deadlocks are no problem here, we have another "global problem". Assigning monotonous increasing unique timestamps to transactions. In a real implementation this could be done by concatenating a local timestamp counter with the node number. Keeping the clocks synchronized is not trivial, but one solution to this problem is discussed by Lomet in [4].

In a distributed database, it is common that more than one scheduler participate in executing a transaction. Because of this, a distributed commit protocol has to be used, to make sure that all participating schedulers reach the same result. Either all perform the commit, or all have to abort. Two well-known protocols are two-phase commit (2PC) and three-phase commit [5]. We have employed 2PC, which the protocol is used by most commercially available distributed database systems.

VI. CONCURRENCY CONTROL IN MOBILE DATABASE SYSTEM

Most of concurrency control strategies are based on three mechanisms viz., locking, timestamps and optimistic concurrency control. Though these schemes are well suited for traditional database applications, they don't work efficiently in mobile environments. Due to various constraints in the mobile environment and nature of different online applications, traditional concurrency control mechanism may not work effectively.

Concurrency control deals with the issues involved in allowing Simultaneous accesses to shared data items. Atomicity, consistency, and isolation of transactions are achieved in the database through concurrency control mechanisms. In particular, mobile applications have to face disconnections. It is expected that the transaction continues when the mobile host is disconnected. Hence there is a need of optimistic replication techniques.

In optimistic replication, shared data is replicated on mobile hosts and users are allowed to continue their work while disconnected. After successful completion of local operations at mobile host, the results are later propagated to fixed hosts. In the earlier approaches whenever a concurrency violation occurs i.e. data items are updated at fixed host the conflicting transaction using the similar data items was aborted. In this approach the conflicting transaction is not aborted but it is restated with new state of the data items.

VII. CONCURRENCY CONTROL IN REAL-TIME DATABASE SYSTEM

The data access scheduling policies are commonly referred as concurrency control protocols. These protocols have the responsibility to ensure that although transactions are executed concurrently with interleaving operations, the committed or certified transactions can be ordered or given a certification time stamp ordering so that the net effect on the database is equivalent to the execution of these transactions in a serialized order one at a time. The general approach for scheduling transactions in soft RTDBS is to use existing techniques in CPU scheduling, buffer management, I/O scheduling, and concurrency control, and by applying time critical scheduling methods to observe the timing requirements of transactions. A lot of research has been conducted in this regard (Agrawal et al, 1992; Son et al, 1995, Abbott and Garcia-Molina, 1989 & 1992; Lin and Son, 1990, Huang et al, 1992 and Idoudi et al, 2009), which have developed and analyzed many paradigms that ensure timing constraints while scheduling transactions.

In this paper CC protocols are for time critical transaction (e.g. with deadline) and specialized CCP for RTDBS. Conventional CCPs include: Pessimistic Concurrency Control (modification of 2PL being used in conventional DBMS) (PCC), Optimistic Concurrency Control (OCC), Timestamp Ordering (TO) and Multiversion

Concurrency Control (MCC). Whereas the specialized one's which we going to survey here includes: Speculative Concurrency Control, Hybrid Algorithms and Real-Time Index Concurrency Control (RICC).

2.1 Pessimistic Concurrency Control (PCC) Protocols

Two Phase locking (2PL) is the most common pessimistic concurrency control protocol in conventional database systems. In 2PL if there is a lock conflict, the requesting transaction is blocked and put into a wait state. Most locking protocols proposed for RTDBS are based on one of the following two approaches: priority abort and priority inheritance. The priority abort approach aborts low priority transactions when priority inversion occurs (Abbott and Garcia-Molina, 1989). On the other hand, the priority inheritance approach allows a low priority transaction to execute at the highest priority of all the higher priority transactions it blocks. Studies have shown better results of priority inheritance than those based on priority abort.

Priority Abort PCC

This algorithm incorporates a conflict resolution scheme that ensures that high priority transactions are not delayed by the low priority transactions. In particular when a transaction T (high priority one) requests a lock on an object held by one or more lower priority transactions in a conflicting mode, the lock-holding transactions are restarted and T is granted the lock. If T's priority is lower than that of any of the lock holders, it awaits for the object to be released (as in standard 2PL). The drawback of this protocol is that transactions may be restarted by higher priority transactions, which are discarded later and these wasted restarts result in performance degradation.

Priority Inheritance PCC

The priority inheritance resolves the priority problem by considering only the actual conflicting transactions (Sha et al, 1987). Whenever a requester blocks behind a lower priority lockholder, the lock holder inherits the priority of the lock requester, until it terminates and releases the lock. Because of the increase in priority the lock-holding transaction may finish sooner, resulting in reduced blocking time for the high priority transaction. Since a high priority transaction requests a lock on an object by a lower priority transaction in conflicting mode. The biggest drawback of this protocol is that the blocking time of high priority transactions is unpredictable in their duration.

VIII. CONCURRENCY CONTROL IN MULTIMEDIA DATABASE

MMDBMS is that it provides simultaneous access to information for many clients via TCP/IP network. The

problems that should be handled refers to process multiple requests and access the same set of data in a concurrent environment. The system must include a synchronization algorithm to ensure that the information doesn't get corrupted when multiple clients' requests access concurrently the same set of data. However, in most of the cases the information is frequently read and only occasionally written. It is far more efficient to allow all reading re-requests to be executed simultaneously and only write requests to be executed in an exclusive manner.

The locking mechanism that was chosen for the system is based on L. Lamport's bakery algorithm. This algorithm was chosen because it offers a good balance between performance and implementation complexity. There are two types of locks used: shared locks used for reading (e.g.: SELECT) and exclusive locks used for writing (e.g.: INSERT). These types of locks are used only at the table level of granularity. There are not defined row-level locks or other locks at a higher level of granularity.

If a SELECT command is retrieved (that implies reading from database), a read-lock will be enabled on the tables (files) involved in the operation. This lock will be active until the tables (files) will no longer be used. It is a non-exclusive lock, meaning that all other reading requests will be permitted, each of them activating their own read-lock.

If an INSERT command or other command that involves writing into database will be received meanwhile, it cannot be executed. No writes are permitted while any read-lock is active. Instead it will be put in a waiting queue for a random period of time. The write operation can be executed only when no other lock is active. After all locks are inactivated for a specific table, the write-lock can be activated. This type of lock is an exclusive one. No other request (read or write) can be accepted while this is active.

When an operation activates a lock, it can include one or several tables. If there is no foreign key defined on the requested table, only one table will be locked. If the table includes foreign keys, all the connected tables will be locked using the same type of lock for all of them.

In order to override the critical section when locks are activated or upgraded, it is used the Lamport's bakery synchronization algorithm. This way it is not possible for two different users to lock accidentally the same resources. When a lock is no longer needed, it will be deactivated directly without using any synchronization algorithm.

The basic idea for the Lamport's bakery algorithm is quite simple. Each user's request receives a serving number when a lock is needed. The holder of the lowest number is the next one that gets access to resources.

The implementation of the algorithm is presented in pseudocode:

```

Algorithm 1. The Bakery Algorithm
waiting[i] <- true;
number[i] <- max(number[0], number[1],
    ..., number[n-1]) + 1;
waiting[i] <- false;
for j <- 0 ... n-1 do {
    while waiting[j] do nothing;
    while (number[j] != 0) and
        (number[j] < number[i])
        do nothing;
}
*ENTER critical section:
    number[i] <- 0;
    * Activate requested lock

```

To implement this algorithm, two lists are used. There is one entry in each list for every lock request. The first array stores the priority number. The other list contains a Boolean value for each request specifying if that request is in line to receive a number. Each entry uniquely identifies the requesting client and locked resources by three values: the user name, the running thread id, and the table name where the lock is needed. The thread id is needed to avoid some deadlocks when a user locks a resource and then suddenly disconnects. The resource's lock will be automatically released because the thread will no longer exist. Even if the client connects again, it will be allocated on a different thread and he will need another lock.

When a new lock request arrives and needs to be enabled, first it sets its Boolean value to true. Then it is assigned the next number available for waiting its turn. After it receives a number, it's no longer waiting so it sets its waiting value to false. Next, the lock request goes through the first list and if there is a request with a lower number, or a request that's waiting for a number, it waits until that request is finished or assigned a higher number. After the lock manager traverses the list it searches for the request with the lowest number in order to be served and activates the lock. After the operation ends, the system automatically calls a "release lock" command. This command will also include information about: user, thread id, and table.

IX. CONCURRENCY CONTROL IN MULTILEVEL SECURED DATABASES (MLS/DB)

Concurrency control is important for MLS/DBs because a covert channel can be easily created through collaboration of multilevel secure transactions in most

traditional concurrency control protocols. In a MLS/DB, the concurrency control protocol must ensure that there are no covert channels between the transactions at different security levels. Traditional concurrency control protocols such as 2PL and Timestamp Ordering protocols are not suitable for MLS/DBs, because when those concurrency control mechanisms are applied to multilevel secure transactions, problems such as covert channel, too much delay or repeated aborts of high security level transactions, and retrieval anomaly [27] can occur. Consequently, concurrency control algorithms for MLS/DB must address the problems originated by the security and availability issues of the MLS/DB. Several protocols have been proposed for concurrency control in MLS/DBMS. Due to the influx of these protocols, we have classified the protocols into following five categories:

Secure Locking Protocol

In the locking-based approaches, in order to prevent timing channels, the executions of transactions at lower security level are never delayed by the actions of a transaction at a higher security level. This can be accomplished by providing a high priority to a low transaction whenever a data conflict occurs between a high transaction and a low transaction.

In [35], Keefe, Tsai and Srivastava examined the security issues and present a formal framework for secure concurrency control in multilevel databases. In this, they have characterized several level of assurance in secure system and show how a scheduler can affect the security in this framework.

A secure locking-based protocol called S2PL was proposed by Jajodia and McCollum [32], which modified the strict two phases locking protocol to covert channel free protocol. In this protocol, a high security level transaction must release its lock on a data item when a low security level transaction requests a write lock on the same data item. When a read lock by a high security level transaction is broken, high security level transaction is to be aborted. Since a low security level transaction is never blocked or restarted by a high security level transaction, this protocol satisfies the secrecy (covert channel free) and integrity requirement, but a malicious low security level transaction may cause a high security level transaction to be aborted repeatedly, resulting in starvation.

McDermott and Jajodia [37] provide a way to reduce the amount of starvation. According to their approach, whenever a high security level transaction prematurely releases its read lock on a low security level data item due to security reasons, it does not abort and roll-back entirely, but holds its write locks on high security level data items, marks the low security level data item in its private workspace as unread and retries reading this data item by entering into a queue. This queue maintains the list of all high security level

transactions waiting for retrieval to read that particular data item and enables the first transaction in the queue to be serviced first. The modified approach, however, does not always produce serializable schedules [33].

Another secure two-phase locking-based protocol (S2PL), is based on a completely different approach was proposed by Son and David [38]. The basic principle behind this S2PL is to try to simulate the execution of conventional 2PL without blocking the actions of low security level transactions by high security level transactions. This is accomplished by providing a new lock type called virtual lock, which is used by low security level transactions that develop conflicts with high security level transactions. The actions corresponding to setting of virtual locks are implemented on private versions of the data item. When the conflicting high security level transaction commits and releases the data item, the virtual lock of the low security level transaction is upgraded to a real lock and the operation is performed on the original data item. To complete this scheme, an additional lock type called dependent virtual lock is required apart from maintaining, for each executing transaction T_i , lists of the active transactions that precede or follow T_i in the serialization order.

Another solution that has been proposed is to allow users to read and write information at multiple classification levels by decomposing the original transaction into multiple sub-transactions, each of which is assigned a single classification level, and all actions performed by sub-transactions obey the Bell-LaPadula properties. However, even in such a scenario, it is impossible to simultaneously guarantee both transaction atomicity and absence of covert channels [30, 36, 39].

Jajodia et al. [33] proposed two secure locking protocols that attempt to detect all cycles in the serialization graph by painting certain transactions and data items accessed by the high security level transactions whose low security level locks are broken and by detecting a cycle at the moment. The first protocol produces pair-wise serializable histories while the second protocol produces serializable histories if the security levels form a total order.

E. Bertino et al. [37] presented an approach to secure concurrency control for transactions in a multilevel secure environment. This approach, which uses single version data items, is based on the use of nested transactions, application-level recovery, and notification-based locking protocols. The notification protocol is based on the use of signal locks. A signal lock is acquired by a transaction whenever it needs to read lower security level data; such a lock does not delay a write lock request by a low security level transaction on the same data item. Hence, timing covert channels arising from synchronization are eliminated. When a data item on which a write lock is acquired by a transaction is modified, all high security level

transactions holding signal locks on that data are notified by the trusted lock manager, and thus may perform recovery actions. To better support recovery activity, transactions are organized according to the nested transaction model extended with specific primitives for supporting the notification protocol. The proposed approach satisfies most of the properties pointed out in Atluri et al. [36], as basic requirements for a secure concurrency control mechanism in a multilevel environment: it avoids starvation and timing channels, and guarantees serializability.

X. CONCURRENCY CONTROL PROTOCOL FOR REPLICATED REAL TIME DATABASES

Even though S2PL (**Static Two Phase Locking**) [41] is a deadlock free mechanism but it slows down the concurrent processing of multiple transactions. This is due to locking of all the data till the end of the commit phase. Also if a higher transaction arrives at a site than executing one then current transaction is aborted and lock is made available to higher priority one. This makes the wastage of resources. Hence we propose here a new mechanism with augmentation of S2PL.

We will use here a term **Healthy Point (HP)** with **Block (B)/Do not Abort (DA)** which means if a cohort reaches its **Healthy Point (HP)** then it will not be aborted against a higher priority transaction at that site. It means **DA** is used here. And if a lower priority transaction demands a lock then it will be blocked against a higher priority executing one. It means **B** is used here.

The proposed mechanism is:

HP of a cohort:

A cohort reaches its HP after sending a PREPARE message to its replica updaters in its execution phase i.e. in first phase of 2PC.

HP of a replica updater:

A replica updater reaches HP after gaining locks on needed data items.

By this mechanism some significant improvements can be noted in S2PL. Since after HP a cohort has a less probability of abortion hence a blocked transaction can borrow data from executing one. It means waiting and executing time of blocked transaction will get reduced which is very needed in a firm RTDBMS. Also by sending PREPARE message to its replica updaters as shown in Figure 3 the waiting time of a cohort between sending PREPARE message to its updaters and receiving COMMIT message from them will get reduced. Hence overall time of execution of transaction will get reduced.

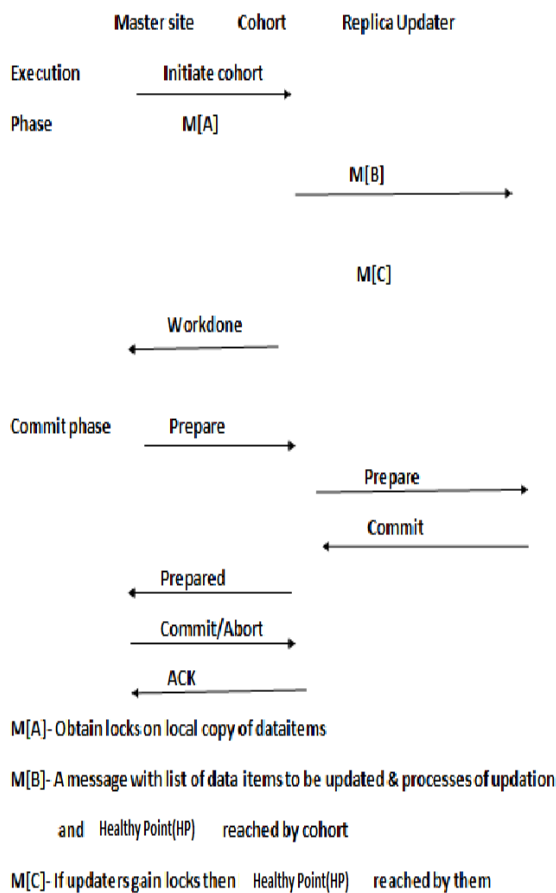
Algorithm:

For a cohort:

```

HP=false;
EXfinish (execution finished) =false;
If (message=INITIATE COHORT) {
  Start execution of cohort;
  EXfinish=true;
}
If (EXfinish=true) {
  Send PREPARE message to its replica updaters;
  HP=true;
  Send WORKDONE message to its coordinator;
}
For a replica updater:
If (message=PREPARE&&lock obtained=true) {
  HP=true;
  After execution send COMMIT message to its cohort;
}

```



Here we present CIRS (Concurrency control In Replicated real-time Systems) a state conscious Concurrency control protocol in replicated distributed environment which is specially for firm real-time database system. CIRS mechanism uses S2PL (Static Two Phase Locking) for deadlock free environment. It also includes veto power given to a cohort after receiving PREPARE message from its coordinator.

XI. CONCLUSIONS

This paper is an attempt to summarize available literature pertaining to work in the direction of developing concurrency control protocols for different database systems. In distributed database system that is considered to be more reliable than centralized database system. It is really important for database to have the ACID properties to perform. In distributed object-oriented database, a scheduler module is available, that is a *black box*, it communicates with the other modules through interface. In the distributed version of the simulator, two-phase locking and timestamp ordering schedulers are implemented. The scheduler receives an operation from the transaction manager, and processes it according to its scheduling technique. When the scheduler decides that an operation can be sent to the data manager, the data manager is called. When the operation has completed, the scheduler will be notified by a call from the data manager.

In a distributed database, it is common that more than one scheduler participate in executing a transaction. Because of this, a distributed commit protocol has to be used, to make sure that all participating schedulers reach the same result. Either all perform the commit, or all have to abort. Two well-known protocols are two-phase commit (2PC) and three-phase commit [5]. We have employed 2PC, which the protocol is used by most commercially available distributed database systems. In Real Time Data Base each transaction is associated with a priority, high priority transaction has earlier deadline. Higher priority transaction obtains the lock first. Lower priority transaction has to sacrifice the lock. If transaction confronts the same priority transaction, then lock is obtained by timestamp ordering.

Concurrency control in Mobile Databases has the same behaviors with those in multidatabase systems in many aspects. Many approaches in multidatabase systems can be extended to mobile multidatabase environment. The differences in Mobile Databases are that transactions in Mobile Databases have mobility and long-lived nature.

In this paper we present CIRS (Concurrency control In Replicated real-time Systems) a state conscious concurrency control protocol in replicated distributed environment which is specially for firm real-time database system. CIRS mechanism uses S2PL (Static Two Phase Locking) for deadlock free environment. It also includes veto power given to a cohort after receiving PREPARE message from its coordinator. Also with some more assumptions like sending an extra message in execution phase but after completion of execution at local copy which is described later in this paper the proposed mechanism has a significant increase in performance over O2PL and MIRROR in decreasing execution time of the current transaction and it also decreases the waiting time of transactions in wait queue.

In Multi-level secure database systems (MLS/DBSs) are shared by concurrent transactions with different clearance levels and manage data objects with different classification levels. We proposed and evaluated a new secure multiversion concurrency control protocol. It was observed that our protocol has better response than SMVCC. In addition to this, results show that our protocol achieves fair performance than SMVCC across different security levels. Here we implemented a new lock that is virtual lock.

REFERENCES

- [1]. Navathe Elmasri, Database Concepts, *Pearson Education*, V edition (2008)
- [2]. Fundamentals of DBMS, Lakhanpal Publisher, III edition (2008)
- [3]. Arun Kumar Yadav & Ajay Agarwal, An Approach for Concurrency Control in Distributed Database System, Vol. 1, No. 1, pp. 137-141, January-June (2010)
- [4]. D. B. Lomet. Consistent Timestamping for Transactions in Distributed Systems. Technical Report CRL 90/3, Digital Equipment Corporation, Cambridge Research Lab, 1990.
- [5]. M. T. O'zsu and P. Valduriez. Principles of Distributed Database Systems. Prentice-Hall, 1991.
- [6]. Salman Abdul Moiz and Dr. Lakshmi Rajamani, "A Real Time Concurrency control in mobile environments using on demand Multicasting", *IJWMN*.
- [7]. Acharya, S., Alonso, R., Franklin, M. and Zdonik, S., "Broadcast disks: data management for asymmetric communication environments", *Proc. ACM SIGMOD 1993 Int. Conf. on Management of Data*, pp. 199-210, 1999.
- [8]. Acharya, S., Franklin, M. and Zdonik, S., "Disseminating Updates on Broadcast Disk", *Proc. 22nd VLDB Conference*, pp. 354-365, 1996.
- [9]. Acharya, S., Franklin, M. and Zdonik, S., "Balancing Push and Pull for Data Broadcast", *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 183-194, 1997.
- [9]. D. Agrawal, A. El Abbadi and R. Jeffers. Using Delayed Commitment in Locking Protocols for Real-Time Databases. In *Proc. of ACM SIGMOD Conference*, June 1992.
- [10]. L. Sha, R. Rajkumar and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. Technical Report CMU-CS-87-181, Depts. of CS, ECE and Statistics, Carnegie Mellon University, 1987.
- [11]. J. Haritsa, M. Carey and M. Livny. Data Access Scheduling in Firm Real-Time Database Systems. In *Journal of Real-Time Systems*, September 1992.
- [12]. L. Sha, R. Rajkumar and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. Technical Report CMU-CS-87-181, Depts. of CS, ECE and Statistics, Carnegie Mellon University, 1987.
- [13]. Haritsa, J., Carey, M., & Livny, M. (1993). Value-based scheduling in real-time database systems. *VLDB Journal*, 2, 117.
- [14]. R. Abbott and H. Garcia-Molina, "Scheduling real-time transactions with disk resident data." In *Proceedings of the 15th international conference on Very large data bases (VLDB '89)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 385-395. 1989.
- [15]. Y. Lin and S.H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proc. Real-Time Systems Symp.*, pp. 104-112, Dec. 1990
- [16]. L. Lamport, A New Solution of Dijkstra's Concurrent Programming Problem, *Communications of the ACM* 17(8), pp. 453-455, 1974.
- [17]. A.S. Tanenbaum, *Modern Operating Systems* (Second Edition), Prentice Hall, 2001.
- [18]. Haritsa, J.R., Carey, M.J., and Livny, M., "Data Access Scheduling in Firm Real-Time Database Systems", *The Journal of Real-Time Systems*, no. 4, 1992, pp. 203-241.
- [19]. Chen, H., & Chin, Y. H. (2003). An adaptive scheduler for distributed real-time database systems. *Information Sciences*, 153, 55. doi:10.1016/S0020-0255(03)00073-2