Performance study of Replica Concurrency Control Algorithms for Distributed Databases

By
Kamal Bista

A dissertation submitted to the
Central Department of Computer Science and Information Technology,
Tribhuvan University
in partial fulfillment of the requirements for the degree of

Master's Degree in Computer Science and Information Technology

TRIBHUVAN UNIVERSITY

Kirtipur, Kathmandu, Nepal

August, 2008

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology

Date: _____

Letter of Recommendation

Mr. ***Kamal Bista*** has carried out this dissertation work entitled **"Performance study of Replica Concurrency Control Algorithms for Distributed Databases"** under my supervision and guidance. This dissertation bears the candidate's own work and is in the form as required by Central Department of Computer Science and Information Technology, Tribhuvan University. I, therefore recommend for further evaluation.

Prof. Dr. Shashidhar Ram Joshi
Department of Electronics and Computer Engineering
Institute of Engineering, Pulchowk Campus, Pulchowk
Tribhuvan University
Supervisor

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology

Letter of Approval

We certify that we have read this dissertation and in our opinion it is satisfactory in the scope and quality as a dissertation in the partial fulfillment for the requirement of Master's degree in Computer Science and Information Technology.

Evaluation Committee

_____	_____
Head, Central Department of Computer Science and Information Technology Tribhuvan University, Nepal	**Prof. Dr. Shashidhar Ram Joshi** Department of Electronics and Computer Engineering Institute of Engineering Tribhuvan University **Supervisor**
_____	_____
Internal Examiner	**External Examiner**

Date: _____

Acknowledgements

It is a great pleasure for me to acknowledge the contributions of a large number of individuals to this work. First of all, I would like to thank my advisor ***Prof. Dr. Shashidhar Ram Joshi*** for giving me an opportunity to work under his supervision and for providing me guidance and support through out this work. I have learned many principles on performing good research from him.

I would like to express my sincere gratitude to ***Prof. Dr. Devi Dutta Paudyal*** (Former Head, Central Department of Computer Science and Information Technology) for his inspiration and encouragement during two years study of my Master's Degree.

I would like to express my gratitude to the respected teachers ***Prof. Dr. Srinath Srinivasa*** (IIIT – Banglore, India), ***Prof. Dr. Laxmi P. Gewali*** (University of Nevada, Las Vegas, USA), ***Prof. Sudarshan Karanjeet***, ***Associate Prof. Manish Pokharel***, ***Asst. Prof. Arun Timilsina, Asst. Prof. Dr. Tanka Nath Dhamala*** (Head, CDCSIT), ***Asst. Prof. Min B. Khati, Asst. Prof. Hemanta Bahadur G.C.*** and all other teacher who have taught us in our Master Degree.

I am in debt to Achyut Pd. Pathak, Hem Raj Aryal and Dinesh Khadka for their fruitful discussions. Last but not least, I would like to thank my family members for their constant support and encouragement.

Kamal Bista

Abstract

This study examines three replica concurrency control algorithms namely Distributed 2PL, Distributed OCC, and Distributed O2PL for distributed database systems. Four Different algorithms are performed are performed to evaluate the performance of above algorithms when they are incorporated with real-time data conflict resolution techniques namely PA, PB, PI, PA_PB. Among the four experiments, first experiment evaluates the performance of the various conflict resolution mechanisms (PA, PB, PI and PA_PB) when integrated with the 2PL and O2PL concurrency control protocols. Experiment 2 evaluates the performance of CC protocols based on the three different techniques: 2PL, O2PL and OCC. Experiment 3 is performed to evaluate the performance of these algorithms under different update frequencies. Experiment 4 is performed to evaluate the performance of these algorithms while varying number of replicas. Results of these experiments are analyzed and presented.

The performance metric employed for all experiments is *MissPercent*, *the percentage of transactions that miss their deadlines.* MissPercent values in the range of 0 to 30 percent are taken to represent system performance under "normal" loads, while MissPercent values in the range of 30 to 100 percent represent system performance under "heavy" loads. Several additional statistics are used to aid in the analysis of the experimental results, including the *abort ratio*, the *message ratio*, *priority inversion ratio (PIR)*, and the *wait ratio*, which is the average number of waits per transaction. Further, the *useful resource utilization* is also measured as the resource utilization made by those transactions that are successfully completed before their deadlines.

All the missed deadline percentage for all experiments in this study is shown by graphs which only consider mean values that have relative half widths about the mean of less than 10% at the 90% confidence interval, with each experiment having been run until at least 10000 transactions are processed by the system.

List of Figures

List of Tables

LIST OF ACRONYMS

2PC	Two Phase Commit
2PL	Two Phase Locking
CM	Cache Manager
CPU	Central Processing Unit
DAR	Data Access Ratio
DM	Data Manager
DRTDBS	Distributed Real Time Database System
O2PL	Optimistic Two Phase Locking
OCC	Optimistic Concurrency Control
PA	Priority Abort
PB	Priority Blocking
PEP	Performance Evaluation Parameters
PI	Priority Inheritence
PIR	Priority Inversion Ratio
RCCPA	Replica Concurrency Control Performance Analyzer
RM	Recovery Manager
ROWA	Read One copy, Write All copies
RTDBS	Real-Time Database System
TM	Transaction Manager

Contents

Chapter 1: Introduction

Many time-critical database applications are inherently distributed in nature. Recent applications include the multitude of directory, data-feed and electronic commerce services that have become available on the World Wide Web. The performance, reliability, and availability of such applications can be significantly enhanced through the replication of data on multiple sites of the distributed network. A pre-requisite for realizing the benefits of replication, however, is the development of efficient replica management mechanisms. In this field, many researchers contribute their knowledge and developed many concurrency control algorithms. Most of these algorithms are based on three basic approaches: locking, timestamps and optimistic concurrency control. Many researchers evaluate the performance of these concurrency control algorithms. Uluosy, O. studied the performances of classical 2PL protocol when augmented with priority abort (PA) and priority inheritance (PI) conflict resolution techniques [1]. However differing with prior performance studies, this study concentrates their efficiency in replicated environment in distributed processing especially of OCC, 2PL, and O2PL. This study evaluates the performances of 2PL and O2PL while these algorithms are incorporated with several data conflict resolution techniques such as PA, PB, PI and state-conscious priority blocking (PA_PB).

This study examines different replica concurrency control algorithms such as standard Distributed 2PL, Distributed OCC, and Distributed O2PL.

The performances of these algorithms are evaluated with different class of transaction and performance is indicated by number of performance parameters: Load, Message Cost, Data Access Ratio, and Update Frequency.

Beside the performance study of replica concurrency control algorithms for distributed databases, different theoretical study of concurrency control algorithms are studied and analyzed.

This study is divided into 7 chapters. Chapter 2 is a foundation for the study of concurrency control in database system. It describes database system components

including transaction manager and scheduler. Moreover, it briefly describes transaction and transaction processing in database system.

Chapter 3 describes the major concurrency control problems: dirty read problem, fuzzy read problem, lost update problem and phantom problem. These problems are generally arises in database management system due to concurrent access in database system. Concurrency control algorithms (i. e. replica concurrency control algorithms in this study) must deal with these problems to ensure database consistency. Moreover, this chapter describes non-recoverability and cascading abort as concurrency control problems.

Chapter 4 describes various replica concurrency control algorithms in detail. Moreover, this chapter describes various Data Conflict Resolution Mechanisms such as Priority Blocking (PB), Priority Abort (PA), Priority Inheritance (PI), OPT-WAIT (for OCC protocol), and State-Conscious Priority Blocking (PA_PB) to deal with data conflict that arises in concurrent execution of transaction.

Chapter 5 describes different performance parameters and experiment strategies to evaluate the performance of replica concurrency control algorithms. This chapter also describes the program model for various experiments.

Chapter 6 presents four different experiments to evaluate the performance of replica concurrency control algorithms. Each experiment evaluates the performance parameters and performance results are summarized.

Chapter 7 summarizes the results of each experiment as conclusions of the study. This chapter clearly expresses the performance of replica concurrency control algorithms in different environments. Moreover this chapter also describes the limitations of this study and explores the direction for further study in the area of replica concurrency control algorithms.

Chapter 2: Foundations for the Study of Concurrency Control

2.1 Introduction

The database system components and transaction processing ([2], [3]) are the foundation for the study of concurrency control ([4], [5], [6]) in database management system. This chapter briefly describes database system components and transaction processing providing the foundation for the study of currency control in database management system.

2.2 Database System Components

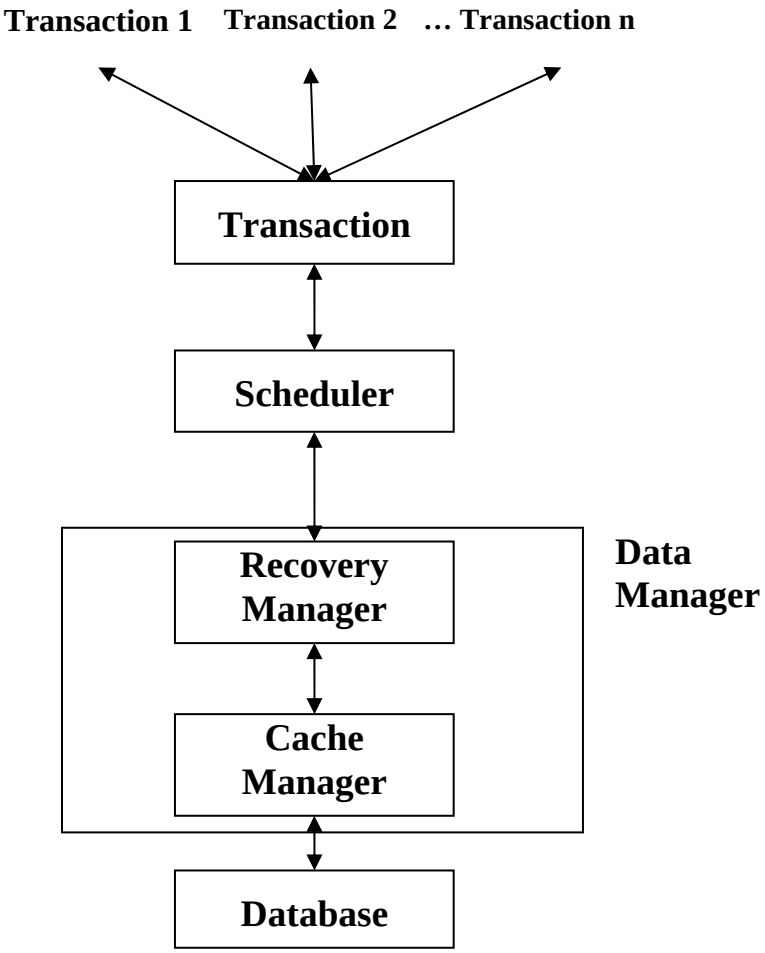

Figure 2.1 Database System Components

In general, database system consists of four components: Transaction Manager(TM), Scheduler, Recovery Manager (RM) and Cache Manager (CM). Transaction Manager is responsible to perform any required preprocessing for database and transaction operations that receives from transaction. Scheduler is major component for concurrency control. It is responsible to control the relative order of database and transaction operations to execute. Recovery Manager (RM) is major component for recovery from failures ([5], [6]). It is responsible to commit and abort the transaction. And finally, Cache Manager (CM) is responsible to actually perform database and transaction operations.

2.2.1 Transaction manager

The major function of transaction manager is to establish the communications between user transaction and database. That is, transaction interacts with the database through a transaction manager (TM). The TM receives database and transaction operations issued by transactions and forwards them to the scheduler. If transaction is aborted, TM is responsible to resubmit the transaction to scheduler. In distributed database system environment [5] TM is more responsible, it has to decide in which site transaction operation has to send for scheduler.

2.2.2 The Scheduler

The scheduler is a primary database system component for concurrency control. Scheduler is responsible to relatively order the execution of database and transaction operations such that resulting execution is serializable [7]. It may also ensure that execution avoids cascading aborts and strict execution [6]. That all depends upon the concurrency control algorithm in which schedule/scheduler is based. In fact, schedule is a program, based on concurrency control algorithms for serializable execution of database and transaction operations.

There are three basic actions scheduler performs once scheduler receives database and transaction's operations from transaction.

(a) Execute: Scheduler pass transaction's operation to Data Manager (DM) to execute. When DM finishes execution of passed operation it informs scheduler. Moreover, if operation is read, it reads a data value from database and it relays back to transaction.

(b) Reject: Scheduler may refuse to process the operation which causes transaction to be aborted. Abort can he issued by transaction or TM.

(c) Delay: Scheduler may delays operation placing it in queue. Later scheduler can either execute or reject it.

These three actions of scheduler are preliminary to control the order of execution of database and transaction's operations. When it receives an operation from the transaction, usually tries to pass it to the DM. If it is unable to execute without producing non-serializable execution, either it delays or reject it. If scheduler finds operation which cannot be correctly processed in further it directly rejects the operation. If scheduler finds possibility to correctly process operation in future it simply delays the operation.

Example 2.2.2.1

Let us consider two transactions

Transaction T_1

```
Procedure DepositA
Begin
  Read1(Accounts[A]);
  Write1(Accounts[A],$100);
  Commit1;
End;
```

Transaction T_2

```
Procedure DepositB
Begin
  Read2(Accounts[A]);
  Write2(Accounts[A],$500);
  Commit2;
End;
```

Consider a possible concurrent schedule produced by T_1 and T_2 as below

T_1	T_2
$Read_1$(Accounts[A]);	
	$Read_2$(Accounts[A]);
	$Write_2$(Accounts[A],$500);
	$Commit_2$;
$Write_1$(Accounts[A],$100);	
$Commit_1$;	

Table 2.1 Non-serializable execution

The above execution is non-serializable. To avoid non- serializable execution, the scheduler might reject $Write_2$, causing transaction T_2 to abort. Transaction manager need to resubmit T_2 during which T_1 may already committed before T2 commit. This maintains serializable execution. Alternatively, the scheduler could delay $Read_1$, until T_2 commits its write. Such scheduling decision can be made using appropriate scheduling algorithms [4].

2.2.3 Recovery Manager

Recovery manager is responsible for restoring the database from most recent consistent state. Recovery manager keeps track of the following operationsin the system log:

- **begin_transaction:** This marks the beginning of transaction execution.
- **read or write:** These specify read or write operations on the database items that are executed as part of a transaction.
- **end_transaction:** This specifies that read and write transaction operations have ended and marks the end limit of transaction execution. At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

- **commit_transaction:** This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- **rollback (or abort):** This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

2.2.4 Cache Manager

Cache manager coordinates buffers of data that store data before writing to the database with database and transactions. Transaction; that performs read operation first seeks the data item in buffers. If not in buffers, then transaction make a trip to database after then it accesses the data item through buffers.

2.3 Centralized and Distributed Database System

2.3.1 Centralized Database System

Centralized database system [5] consist a single database unit and it is placed in a single computer system. It basically adopts client server environment. In client server environment, database is place in server and number of clients may connect to central database stored in server via communication network.

For a centralized database system, centralized computer system could be the underlying computer system on which it runs. In general, centralized database system consists of a central processor, some main memory, secondary storage devices, and I/O devices. It may also come with multiprocessors in which each processor has direct access to all of main memory and to all I/O devices.

2.3.2 Distributed Database System

Distributed database system is a collection of sites connected by communication network. Each site in distributed database is centralized database system which stores a copy of the entire database. So the components of distributed database are same as for centralized

database system: Transaction Manager (TM), Scheduler, Recovery Manager (RM), and Cache manager (CM).

Since database is distributed over several sites, each transaction may consist of one or more processes that need to execute at one or more sites. TM needs to forward each operation to appropriate scheduler in which site where data arc available to process the operation. TM can communicate with all scheduler exist in all sites via communication network.

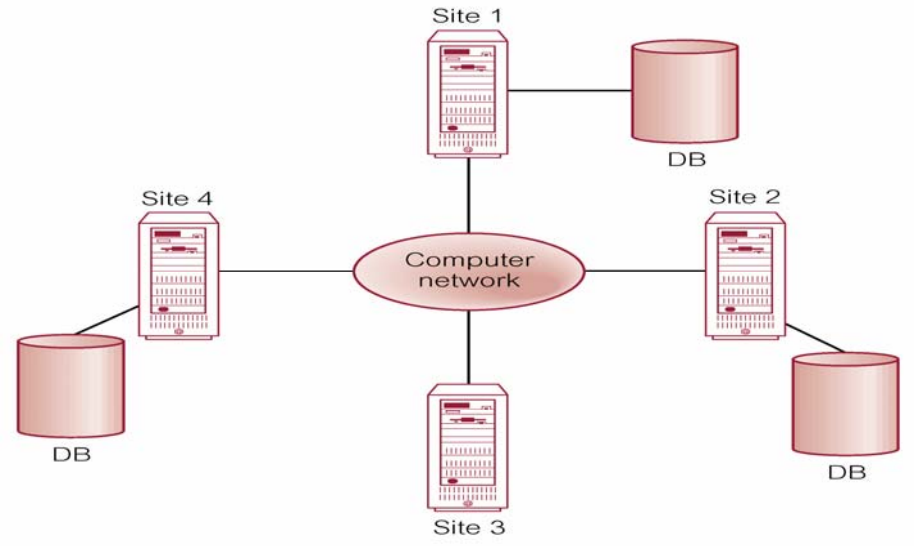

Figure 2.2 Distributed Database Systems [8]

2.4 Transaction Processing

2.4.1 Motivation

Concurrency is a mandatory property of a database system it must allow by the database system. In concurrent environment, read and write operations of one database user may interfere with other. Due to interference only some read/write operations of database user may execute rest of read /write operations could not be executed since database system assumes each read/write operation as individual and independent task. If all read and write operations issued by database user are really independent in nature, partial execution of read/write operations does not create big problem. But in reality, each database read or write operation rarely represent a complete task of database user. In such

situation, it may lead inconsistency problem [9]. This really demands encapsulation of set of database operations which can perform a complete task. In fact, transaction is initiated with this concept. It isolates set of database operations providing set of operations as a single unit. If any one of the operations that exist in set of database operation could not execute either because of concurrent transaction interfere or because of failure, database system ignores set of all operations that exist. This helps to ensure consistency of database in concurrent environment. That is the major motivation of transaction is to ensure consistency allowing concurrent execution.

2.4.2 Definition of Transaction

A transaction is a unit of program consisting set of database operations whose execution may change the database state. If database is initially in consistent state before executing transaction, database should remain in consistent state at the end of transaction. To ensure consistency of database before and after execution of transaction, it needs to be atomic [10]. Read, Write, Commit, and Abort are major database operations that exist in transaction.

Transaction can also be defined as a collection of actions that make consistent transformations preserving database consistency.

Example 2.4.2.1: Fund transfer from account A to account B

```
Procedure FundTransfer
Begin
  Input(A,B );
  temp = Read(Accounts[A]);
  temp1 = temp1-$100
  write( Accounts [A],temp1);
  temp2 = Read(Accounts[B]);
  temp2 = temp2 +$100
  write( Accounts[B],temp2);
  commit;

End;
```

This can be expressed as

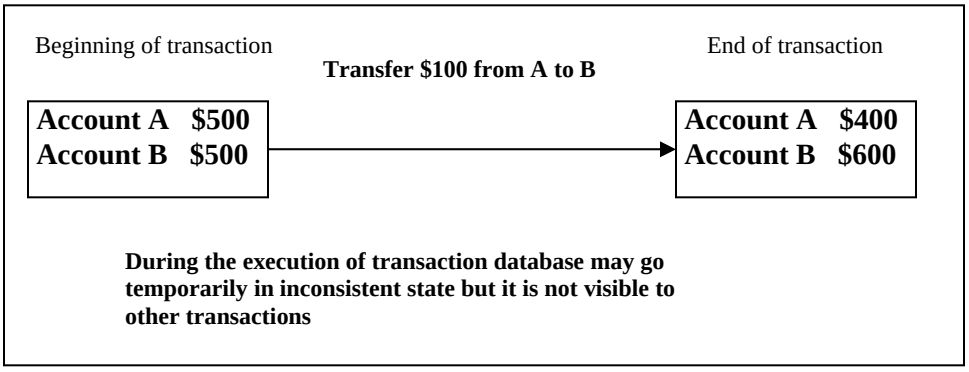

Figure 2.3 Transaction execution instance

2.4.3 Transaction Properties

The definition of transaction tells states of transaction and its actions are not visible to other transactions or database users until transaction terminates. That is, partial changes made by transaction are not visible outside this transaction. Only when transaction terminates, database users notified its success or failure and changes made by transaction are made visible. We already discussed that these characteristics are foundation for, currency control. To achieve these characteristics, transaction should have atomicity, consistency, isolation and durability properties, called ACID properties ([2], [10]) of transaction.

The atomicity property of transaction tells transaction is an individual unit. It needs to execute set of all operations that belong to this transaction then only system can reflects changes made by this transaction. This is helpful to modify/update database in consistent manner. Let us consider task, which is responsible to transfer funds from account A to B. Assume that, failure occurs power failure or hardware failure or software error) immediately after account A is updated but before update perform in account B. Definitely, such incomplete transaction leads database in inconsistent state [9]; such incomplete execution of transaction's effect should wipe out. Transaction's atomicity property does not allow violating such integrity [9]. Transaction manager (TM) is responsible for ensuring atomicity property of transaction.

The consistency property of transaction ensures transaction should preserve consistency of database during its execution. That is, if database was initially in consistent state before start of transaction execution, then database should again in consistent state once transaction terminates. Database user itself is responsible to ensure consistency property of transaction. In fund transfer transaction, we could enforce consistency criteria as sum of amount of all account must not be changed by fund transfer transaction.

The isolation property of transaction tells actions performed by transaction should be isolated or hidden from outside the transaction until transaction is not terminated. That is, even though transactions are running concurrently, any changes made by transaction is not visible to other transitions or database user until transaction is not terminated. For example transaction T_1 is executing fund transfer transaction form account A to B and another transaction T_2 try to read sum of the amount from account A and B. In such case, isolation property of transaction does not allow to read changes made by transaction T_1 to T_2 until and unless T_1 is not terminated. The database system component scheduler is responsible for ensuring isolation property of transaction.

The durability property of transaction ensures committed actions of transaction must reflect in database. Any failure, after transaction commit will not cause loss of updates made by this transaction. The recovery manager is responsible for ensuring durability property of transaction. A simple idea for ensuring durability property of transaction is to keep the log of all changes carried out before writing the effect of updated transaction to disk. The content of log can be used by TM to restore the database state during the system failure or system restart.

2.4.4 State of Transaction

Transaction model consists of the following state of transactions:

Active: Initial state, transaction stay in this state while it is executing.

Partially-Committed: Transaction stay in this state just after it executes final statement of the transaction. It indicates that it is at the end of transaction. At this point, the

transaction completed its execution but still it may abort because up to this state actual output of transaction may still temporarily residing in main memory; hardware failure may cause impossible to reflect changes made by transaction from main memory to database.

Failed: A transaction is said to be in failed sate once normal execution of transaction can no longer proceed. It could be because of hardware and logical error. Failed transaction must be rollback. That is, all changes made by transaction to database must be undone.

Committed: Signals successful end of transaction. Any changes made by transaction can be safely committed to database which cannot undo in future.

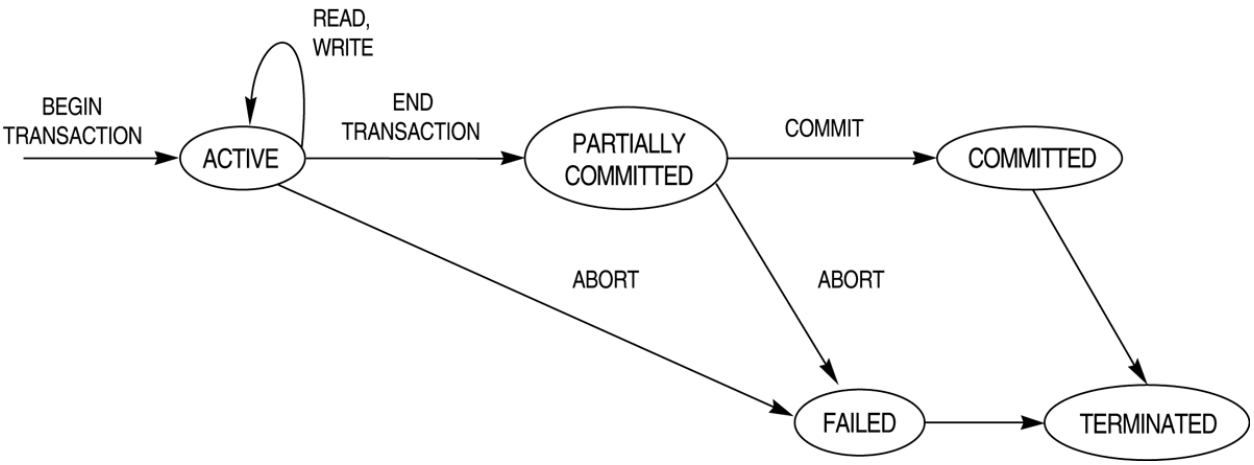

Figure 2.4 Transaction execution states

Once a transaction successfully commits then the database system must guarantee that its updates permanently store in the database, even system crash occurs in the very next moment. It is possible that system may crash just after we issued a COMMIT but before issuing updates to physically write changes in database [11]. It might still be waiting a main memory buffer. The system's restart procedure may store those updates in the database. The general technique to recover from such system crash is to maintain log of each transaction actions, known as log-based recovery [12]. Write-ahead log rule tells, log must be physically written before COMMIT processing completes. Database system should have capabilities to recover from failure.

Chapter 3: Concurrency Control Problems

3.1 Introduction

Generally, database system allows multiple transactions to run concurrently. Concurrent execution of transaction in database system improves database system performance [13], reducing transaction waiting time to proceed. It improves resource utilization. But it may lead the database in inconsistent state due to interference among actions of concurrent transactions. Concurrent execution of transaction in database system leads several concurrency control problems [9]. Major concurrency control problems that may generally arise in concurrent execution will discuss in this chapter.

3.2 Concurrency Control Problems

The main reason of concurrency control problem is interference [9]. In concurrent execution, transaction need to execute in interleave fashion and when number of concurrent transactions executes in interleave fashion there is possibility of interference which may lead different concurrency control problems.

3.2.1 Dirty Read Problem

Suppose transaction T_1 modifies a data item x and another transaction T_2 then reads that data item x before T_1 perform commit or abort. Now, if T_1 perform abort then data item x read by T_2 can never committed. In such case, data item x read by T_2 is known as dirty read. Let us examine a concurrent schedule that demonstrates a possible dirty read problem.

$\mathbf{T_1}$	$\mathbf{T_2}$
$Write_1(x)$	
	$Read_2(x)$
	$Write_2(y)$
$Abort_1$	

Table 3.1 Schedule illustrating dirty read problem

Since T_2 read x (dirty read) that was already written by T_1 but not committed yet T_1 aborts also cause T_2 to be abort, changes made by T_2 is never committed. Dirty read problem in concurrent execution occur if transaction T reads uncommitted transaction and subsequently aborts before T's commit [11]. Similar case is shown in above schedule.

3.2.2 Non - Repeatable (Fuzzy) Read Problem

Suppose transaction T_1 reads x and then another transaction T_2 modifies it. But still T_1 assumes data value of x is unchanged and proceeds further without reading updated values of x. If transaction T_1 need to perform further actions based on x that is updated after T_1's read then it could lead problem in there execution. The same scenario is demonstrated by the following concurrent schedule.

T_1	T_2
$Read_1(x)$	
	$Write_2(x)$
	$Commit_2$
$Read_1(y)$	
$Write_1(x + y \rightarrow z)$	
$Commit_1$	

Table 3.2 Schedule illustrating fuzzy read problem

Here job of transactions T_1 is to read current values of x and y then stores their sum in z. Before T_1's write, data value x read by T_1 is already updated by T_2. But T_1 still assumes old value of x as a current value that T_1 reads before T_2 update it. Therefore, when T_1 commits it cannot write appropriate sum of x and y to z. Here problem occurs due to fuzzy read of T_1 [11].

3.2.3 Lost update Problem

If two or more transaction modifies data item x at a time then lost update problem occurs, update made by one transaction may overwrite by other transaction update. If T_1 updates

x but not committed yet, before T_1 commit, if another transaction T_2, update x and eventually T_2 commits before T_1 then update made by T_1 is lost. Let us consider a schedule which demonstrates lost update problem.

T_1	**T_2**
$Read_1(x)$	
	$Read_2(x)$
$Write_1(x)$	
	$Write_2(x)$
	$Commit_2$

Table 3.3 Schedule illustrating lost update problem

Initially T_1 and T_2 read same data value of x. when T_2 commits, T_1's write to x is overwritten by T_2's write to x.

3.2.4 Phantom Problem

Most of the databases are dynamic; meaning is that there are no fixed numbers of records in which we always perform query to update and to retrieve required data. In normal, we need to add, remove or moved data within database. Such database is called dynamic database [5]. In dynamic database phantom problem may arise.

Suppose transaction T_1 reads a set of data item satisfying some search condition and then another transaction T_2 say creates new data items satisfying the same search condition of T_1 then if T_1 repeats its reads with the same search condition, it will get a set of data items differ from the first read. This problem is known as phantom problem.

Let us consider a concurrent schedule that demonstrate phantom problem in dynamic database. Suppose T_1 is responsible to read data values of x and y then need to store their sum in z and T_2 is responsible to delete the data item x. Assume possible concurrent schedule with T_1 and T_2 as below.

T_1	T_2
$Read_1(x)$	
	$Delete_2(x)$
	$Commit_2$
$Read_1(y)$	
$Write_1(x + y \rightarrow z)$	
$Commit_1$	

Table 3.4 Schedule illustrating phantom problem

Initially T_1 reads data value of x and keep it to add with data value of y but immediately T_2 deletes data item x before T_1 store sum of x and y to z. When T_1 commits T_1 stores sum of x and y even data item x is no longer exist in database.

3.3 Non Recoverability and Cascading Aborts as a Concurrency Control Problem

When transaction T aborts, database system must undo its effects for each data item updated by T. That is, database system need to rollback T's effect from database during abort of T. There are two possible effects of transaction T. T may effect on data value written in the database or it may also affect on other transactions. In both case, aborted transaction's effects should undo from database. If aborted transaction may trigger further abortion, it is known as cascading abort.

Let us consider a schedule which illustrates cascading abort. Assume x and y are data items having initial data value 1 for both x and y.

T_1	T_2
$Write_1(x,2)$	
	$Read_2(x)$
	$Write_2(y,3)$
$Abort_1$	

Table 3.5 Schedule illustrate cascading abort

In the above schedule, when transaction T_1 aborts database system must restore update made by T_1. That is, database system must undo $Write_1$ (x, 2) restoring x=1. Restoring the update made by T_1 is not sufficient since T_2 reads value of x written by T_1, T_2 also need to abort. That is, database system need to undo $Write_2$(y, 3) restoring y=1. Even cascading abort maintain consistency of database by aborting series of transactions, it is in fact a concurrency problem. Cascading abort is really unpleasant. This is considered as a concurrency problem because it requires significant bookkeeping to track which transactions reads from which others, single transaction abortion force to abort one or more other transactions; which is very expensive.

Cascading abort is not always possible. Durability property of transaction tells once a transaction is committed, the database system must guarantee it could not be abort. There would be a situation that where cascading abort required but not possible. This usually happens if transaction T_j reads changes made by other Transaction T_i and T_i aborts after T_j's commit. Let's examine such situation by the following schedule.

T_1	T_2
$Write_1(x, 2)$	
	$Read_2(x)$
	$Write_2(y, 3)$
$Read_1(y)$	$Commit_2$
$Abort_1$	

Table 3.6 Schedule illustrate non recoverable schedule

Here, once transaction T_1 aborts, T_2 need be aborted but it violates durability property of

transaction. T_2 is already committed before T_1 aborted. So here, cascading abort is not possible. Here schedule demands cascading abort but it is not possible, such schedule called non-recoverable schedule. Non-recoverability is in fact concurrency problem and recoverability is required properly for concurrency control. Non-recoverable execution is more danger than cascading abort [9]. Cascading abort is expensive but it does not violate transaction property (durability property of transaction).

Formally, recoverable schedule is defined as follows.
Suppose transaction Tj reads x that was written by transaction Ti in the execution then schedule S is called recoverable if it follows following conditions

- T_j reads x after T_i has written into it.
- T_i does not abort before T_j reads x and
- Every transaction (if any) that write x between T_i writes x and T_j reads x, aborts before T_j read x.

It indicates that, for recoverable execution if T_j reads from T_i then T_j must follow T_i's commit. An execution is recoverable if database system always able to reverse the effects of aborted transaction on other transactions [6]. Recoverability is required to ensure aborting transaction does not change the semantics of committed transaction's operations.

3.4 Avoiding Cascading Aborts and Ensuring Recoverability

It has been already stated that cascading abort and non-recoverable execution are concurrency problems [6], it should avoid during concurrent execution.

Cascading aborts can avoid if database ensures that every transaction read only those data values that were written by committed transactions. To achieve cascadelessness, database system need to delay each Read(x) until transaction that has previously issued a Write (x, val) have either aborted or committed. Avoiding cascading abort also ensures recoverability but enforcing recoverability does not remove the possibility of cascading aborts. Let us reexamine the schedule defined in the table below.

T1	T_2
Write1(x, 2)	
	Read2(x)
	Write2(y, 3)
	Commit2
Abort1	

Table 3.7 Schedule illustrate cascading aborts schedule

Here, this schedule is not cascadelessness and not recoverable. To achieve cascadelessness $Read_2(x)$ must wait till T_1's abort. And definitely it ensures recoverability as well as cascadelessness. In the above schedule if T_1 aborts just before T_2's commits, then schedule becomes recoverable but it does not avoid cascading aborts, abortion of T_1 lead T_2 to abort.

3.5 Strict Execution

From the practical point of view, avoiding cascading aborts is not always enough [5]. A further restriction on execution is often desirable. The cascadelessness schedule only enforces transaction could not read data item x that was already written by uncommitted transaction. But it does not enforce transaction could not write x that was already written by uncommitted transaction. Let us examine the cascadelessness schedule it can lead problem in concurrent execution.

T1	T2
$Write_1(x, 2)$	
	Write2(x, 3)
$Abort_1$	

Table 3.8 Cascadelessness schedule

Here, T2 didn't read x that already written by T1 so schedule is cascadelessness.

According to the definition of cascadelessness schedule, it does not need to enforce T_2 to abort but T_2's write may dependent to T_1's write. If so such cascadelessness schedule may cause problem. Let's look the scenario more preciously, assume that initial value of x is 50. Transaction T_1 is responsible to add 20 in x and transaction T_2 is responsible to add say 5% of current value of x then T_2's write becomes logically invalid when T_1 aborts but T_1 does not need to enforce T2 to abort.

The strict execution is serious about such problem. Strict execution delays T_2's write to x until T_1 abort or commit. That is, strict execution restricts both reads and writes to x if x is already written by T_1 until T_1 is either committed or aborted. Strict execution ensures both cascadelessness and recoverability [6].

Chapter 4: Replica Concurrency Control Protocols

4.1 Introduction

Distributed database systems are multi-user systems, which allow the number of transactions from the different sites to access the same database simultaneously. Concurrent access to shared database may lead database in inconsistent state. To preserve the consistency of database, the database system must adopt some concurrency control mechanisms to ensure that the modifications made by transactions are not lost.

4.2 Replica Concurrency Control Algorithms

To preserve the consistency of database, the database system must adopt some concurrency control mechanisms to ensure that the modifications made by transactions are not lost. For this purpose, here, this study mainly focuses three classical families of distributed Concurrency Control (CC) protocols, Two Phase Locking (2PL), Optimistic Concurrency Control (OCC), and Optimistic Two-Phase Locking (O2PL). All three protocol classes belong to the ROWA ("read one copy, write all copies") category with respect to their treatment of replicated data.

4.2.1 Distributed Two-Phase Locking (2PL)

In the distributed two-phase locking algorithm [15], a transaction that intends to read a data item has to only set a read lock on *any* copy of the item; to update an item, however, 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 by a local cohort and its remote updaters. Only the data locked by a cohort is updated in the data processing phase of a transaction. Remote copies locked by updaters are updated after those updaters have received copies of the relevant updates with the *PREPARE* message during the first phase of the commit protocol. Read locks are held until the transaction has entered the prepared

state while write locks are held until they are committed or aborted.

4.2.2 Distributed Optimistic Concurrency Control (OCC)

Distributed optimistic concurrency control algorithm, OCC [16], extends the implementation strategy for centralized OCC algorithms proposed in to handle data distribution and replication.

In OCC, transactions execute in three phases: *read*, *validation*, and *write*. In the read phase, cohorts only access data items in their local sites and all updating of replicas is deferred to the end of transaction, that is, to the commit processing phase. More specifically, the two-phase commit (2PC) protocol is "overloaded" to perform validation in its first phase, and then installation of the private updates of successfully validated transactions in its second phase.

The validation process works as follows: After receiving a PREPARE message from its master, a cohort initiates *local* validation. If a cohort fails during validation, it sends an ABORT message to its master. Otherwise, it sends PREPARE messages as well as copies of the relevant updates to all the sites that store copies of its updated data items. Each site which receives a PREPARE message from the cohort initiates an updater to update the data in its local work area used by OCC. When the updates are done, the updater performs local validation and sends a PREPARED message to its cohort. After the cohort collects PREPARED messages from all its updaters, it sends a PREPARED message to the master. If the master receives PREPARED messages from all its cohorts, the transaction is successfully *globally* validated and the master then issues COMMIT messages to all the cohorts.

A cohort that receives a COMMIT message enters the write phase (the third phase) of the OCC algorithm. After it finishes the write phase, it sends a COMMIT message to all its updaters which then complete their write phase in the same manner as the cohort.

For the implementation of the validation test itself, an efficient strategy called *Lock-based Distributed Validation* is employed.

An important point to note here is that in contrast to centralized databases where transactions that validate successfully always commit, a distributed transaction that gets locally validated might be aborted later because it fails during global validation. This can lead to wasteful aborts of transactions – other transactions could be aborted when a transaction gets locally validated, but the locally validated transaction itself is aborted later. This is a potential performance drawback for OCC in distributed systems.

4.2.3 Distributed Optimistic Two-Phase Locking (O2PL)

The O2PL algorithm[16] can be thought of as a hybrid occupying the middle ground between 2PL and OCC. Specifically, O2PL handles read requests in the same way that 2PL does; in fact, 2PL and O2PL are *identical* in the absence of replication. However, O2PL handles replicated data optimistically. When a cohort updates a replicated data item, it requests a write lock immediately on the local copy of the item. But it defers requesting write locks on any of the remote copies until the beginning of the commit phase is reached. As in the OCC algorithm, replica updaters are initiated by cohorts in the commit phase. Thus, communication with the remote copy site is accomplished by simply passing update information in the PREPARE message of the commit protocol. In particular, the PREPARE message sent by a cohort to its remote updaters includes a list of items to be updated, and each remote updater must obtain write locks on these copies before it can act on the PREPARE request. Since O2PL waits until the end of a transaction to obtain write locks on copies, both blocking and abort are possible rather late in the execution of a transaction. In particular, if two transactions at different sites have updated different copies of a common data item, one of the transactions has to be aborted eventually after the conflict is detected. In this case, the lower priority transaction is usually chosen for abort in RTDBS.

4.2.4 Time of Updates to Replicas

It is important to note that the *time* at which the remote update processes are invoked is a function of the choice of CC protocol. In 2PL, a cohort invokes its remote replica update processes to obtain locks *before* the cohort updates a local data item in the transaction

execution phase. Replicas are updated during the commitment of the transaction. However, in the O2PL and OCC protocols, a cohort invokes the remote replica update processes only in the *first phase* of the two-phase commit protocol.

4.3 Data Conflict Resolution Mechanisms

Here, the integration of real time cognizant data conflict resolution mechanism into the replica concurrency control protocols is discussed. There are three different ways to introduce real-time associated priorities into locking protocols:

4.3.1 Priority Blocking (PB)

This mechanism is similar to the conventional locking protocol in that a transaction is always blocked when it encounters a lock conflict and can only get the lock after the lock is released. The *lock request queue*, however, is ordered by transaction priority.

4.3.2 Priority Abort (PA)

This scheme attempts to resolve all data conflicts in favor of high-priority transactions. Specifically, at the time of a data lock conflict, if the lock holding cohort (updater) has higher priority than the priority of the cohort (updater) that is requesting the lock, the requester is blocked. Otherwise, the lock holding cohort (updater) is aborted and the lock is granted to the requester. Upon the abort of a cohort (updater), a message is sent to the master (cohort) of the cohort (updater) to abort and then restart the whole transaction (if its deadline has not expired by this time).

The only exception to the above policy is when the low priority cohort (updater) has already reached the PREPARED state at the time of the data conflict. In this case, it cannot be aborted unilaterally since its destiny can only be decided by its master and therefore the high priority transaction is forced to wait for the commit processing to be completed.

4.3.3 Priority Inheritance (PI)

In this scheme, whenever data conflict occurs the requester is inserted into the lock request queue which is ordered by priority. If the requester's priority is higher than that of any of the current lock holders, then these low priority cohort(s) holding the lock subsequently execute at the priority of the requester, that is, they "inherit" this priority. This means that lock holders always execute either at their own priority or at the priority of the highest priority cohort waiting for the lock, whichever is greater.

The implementation of priority inheritance in distributed databases is not trivial. For example, whenever a cohort inherits a priority, it has to notify its master about the inherited priority. The master propagates this information to all the sibling cohorts of the transaction. This means that the dissemination of inheritance information to cohorts takes time and effort and significantly adds to the complexity of the system implementation.

For the optimistic protocol, OCC, the *OPT-WAIT* [18] conflict resolution mechanism is used, described below:

4.3.4 OPT-WAIT

In this mechanism, a transaction that reaches validation and finds higher priority transactions in its conflict set is "put on the shelf", that is, it is made to wait and not allowed to commit immediately. This gives the higher priority transactions a chance to make their deadlines first. After all conflicting higher priority transactions leave the conflict set, either due to committing or due to aborting, the on-the-shelf waiter is allowed to commit. Note that a waiting transaction might be restarted due to the commit of one of the conflicting higher priority transactions.

4.3.5 State-Conscious Priority Blocking (PA_PB)

To resolve a conflict in O2PL, the CC manager uses Priority Abort (PA) mechanism if the lock holder has not passed a point called the *demarcation point*; otherwise it uses PB (Priority Blocking) mechanism.

The demarcation points of a cohort/updater T_i is assigned as follows:

- T_i **is a cohort:**

  when T_i receives a PREPARE message from its master.

- T_i **is a replica updater:**

  when T_i has acquired all the local write locks

Essentially, this study sets the demarcation point in such a way that, beyond that point, the cohort or the updater does not incur any locally induced waits. So, in the case of O2PL, a cohort reaches its demarcation point when it receives a PREPARE message from its master. This happens before the cohort sends PREPARE messages to its remote updaters. It is worth noting that, to a cohort, the difference between PA and PA_PB is with regard to when the cohort reaches the point after which it cannot be aborted by lock conflict. In case of the classical priority abort (PA) mechanism, a cohort enters the PREPARED state after it votes for COMMIT, and a PREPARED cohort cannot be aborted unilaterally. This happens *after* all the remote updaters of the cohort vote to COMMIT. On the other hand, in the PA_PB mechanism, a cohort reaches its demarcation point *before* it sends PREPARE messages to its remote updaters. PA and PA_PB become identical if databases are not replicated. Thus, in state-conscious protocols, cohorts or updaters reach demarcation points only after the two phase commit protocol starts. This means that a cohort/updater cannot reach its demarcation point unless it has acquired all the locks. Note also that a cohort/updater that reaches its demarcation point may still be aborted due to write lock conflict.

4.4 Incorporating PA_PB into the 2PL

PA_PB conflict resolution mechanism which was discussed above in the context of the O2PL, can be also added to the distributed 2PL. For 2PL, we assign the demarcation points of a cohort/updater T_i is assigned as follows:

- T_i ***is a cohort:***

  when Ti receives a PREPARE message from its master

- T_i ***is a replica updater:***

  when T_i receives a PREPARE message from its cohort

One special effect in combining with 2PL, unlike the combination with O2PL, is that a low priority transaction which has reached its demarcation point and has blocked a high priority transaction will not suffer any lock based waits.

4.5 Choice of Post-Demarcation Conflict Resolution Mechanism

In the above description, Priority Blocking (PB) is used for the post-demarcation conflict resolution mechanism. Alternatively, Priority Inheritance could be used instead, as given below:

4.5.1 State-Conscious Priority Inheritance (PA_PI)

To resolve a conflict, the CC manager uses PA if the lock holder has not passed the demarcation point, otherwise it uses PI.

At first glance, the above approach may appear to be significantly better than PA_PB since it does not only prevent close-to-completion transactions from being aborted, but also helps them complete quicker, thereby reduces the waiting time of the high-priority transactions blocked by such transactions.

Chapter 5: Performance Evaluation Strategies

5.1 Performance Parameters

This study identified load, message cost, data access ratio (DAR) and update frequency as performance parameters to evaluate the performance of replica concurrency control protocols.

5.2 Experiment Strategies

To evaluate the performance of the concurrency control protocols described in Chapter 5, a detailed performance evaluation model of a distributed real-time database system (DRTDBS) is developed. This model is based on the distributed database model presented in [17]. A summary of the parameters used in the simulation model are presented in Table 5.

Parameter	**Meaning**	**Setting**
NumSites	Number of sites	4
DBSize	Number of Pages in the databases	1000 pages
ReplDegree	Degree of Replication	4
NumCPUs	Number of CPUs per site	2
NumDataDisks	Number of data disks per site	4
NumLogDisks	Number of log disks per site	1
BufHitRatio	Buffer hit ratio on a site	0.1
ArrivalRate	Transaction arrival rate (Trans./Second)	Varied
SlackFactor	Slack factor in deadline assignment	6.0
TransSize	No. of pages accessed per trans.	16 pages
UpdateFreq	Update frequency	0.25
PageCPU time	CPU page processing	10 ms
InitWriteCPU	Time to initiate a disk write	2 ms
PageDisk	Disk page access time	20 ms
LogDisk	Log force time	5 ms
MsgCPU	CPU message send/receive time	1 ms

Table 5: Performance Evaluation Model Parameters and Default Settings

The database is modeled as a collection of *DBSize* pages that are distributed over *NumSites* sites. The number of replicas of each page, that is, the “replication degree”, is

determined by the *ReplDegree* parameter. The physical resources at each site consist of *NumCPUs* CPUs, *NumDataDisks* data disks and *NumLogDisks* log disks. At each site, there is a single common queue for the CPUs and the scheduling policy is preemptive Highest-Priority-First. Each of the disks has its own queue and is scheduled according to a Head-Of-Line policy, with the request queue being ordered by transaction priority. The *PageCPU* and *PageDisk* parameters capture the CPU and disk processing times per data page, respectively. The parameter *InitWriteCPU* models the CPU overhead associated with initiating a disk write for an updated page.

When a transaction makes a request for accessing a data page, the data page may be found in the buffer pool, or it may have to be accessed from the disk. The *BufHitRatio* parameter gives the probability of finding a requested page already resident in the buffer pool.

The communication network is simply modeled as a switch that routes messages and the CPU overhead of message transfer is taken into account at both the sending and receiving sites and its value is determined by *theMsgCPU* parameter – the network delays are subsumed in this parameter. This means that there are two classes of CPU requests – local data processing requests and message processing requests. Any distinction is not made, however, between these different types of requests and it is only ensured that all requests are served in priority order.

With regard to logging costs, we explicitly model only *forced* log writes since they are done synchronously, i.e., operations of the transaction are suspended during the associated disk writing period. This logging cost is captured by the *LogDisk* parameter.

Transactions arrive in a Poisson stream with rate *ArrivalRate,* and each transaction has an associated firm deadline, assigned as described below. Each transaction randomly chooses a site in the system to be the site where the transaction originates and then forks off cohorts at all the sites where it has to access data. Transactions in a distributed system can execute in either *sequential* or *parallel* fashion. The distinction is that cohorts in a sequential transaction execute one after another, whereas cohorts in a parallel transaction

are started together and execute independently until commit processing is initiated. However, only sequential transactions are considered in this study. However, it is noted that the execution of replica updaters belonging to the same cohort is *always in parallel.*
The total number of pages accessed by a transaction, ignoring replicas, varies uniformly between 0.5 and 1.5 times *TransSize*. These pages are chosen uniformly (without replacement) from the entire database. The proportion of accessed pages that are also updated is determined by *UpdateF req*.

Upon arrival, each transaction T is assigned a firm completion deadline using the formula

$$Deadline_T = ArrivalTime_T + SlackFactor * R_T$$

where $Deadline_T$, $ArrivalTime_T$, and R_T are the deadline, arrival time, and resource time, respectively, of transaction T, while *SlackFactor* is a slack factor that provides control of the tightness/slackness of transaction deadlines. The resource time is the total service time at the resources at all sites that the transaction requires for its execution *in the absence of data replication*. This is done because the replica-related cost differs from one CC protocol to another.
It is important to note that while transaction resource requirements are used in assigning transaction deadlines, *the system itself lacks any knowledge of these requirements* in our model since for many applications it is unrealistic to expect such knowledge [18]. This also implies that a transaction is detected as being late only when it *actually* misses its deadline.

As discussed earlier, transactions in an RTDBS are typically assigned priorities so as to minimize the number of killed transactions. In our model, all cohorts inherit their parent transaction's priority. Messages also retain their sending transaction's priority. The transaction priority assignment used in all of the experiments described here is the widely-used *Earliest Deadline policy* [20], wherein transactions with earlier deadlines have higher priority than transactions with later deadlines.

Deadlock is possible with some of the CC protocols that we evaluate in our experiments, deadlocks are detected using a time out mechanism. Both this study's own model as well as the results reported in previous studies [21] show that the frequency of deadlocks is extremely small – therefore a low-overhead solution like timeout is preferable compared to more expensive graph-based techniques.

5.3 Program Overview

The simulator Program "Replica Concurrency Control Performance Analyzer" (RCCPA) is designed to evaluate the performance of Distributed 2PL, O2PL, and OCC. Simulation program RCCPA allows different experiments.

Experiments Component of RCCPA allows to choose the particular experiment. Performance Evaluation Parameter (PEP) component allows us to set different parameters for each experiment. Performance Report component of RCCA analyze the performance reports of each experiments and performance trends of locking algorithms are present in graphical representation.

5.4 Snapshot of program Components

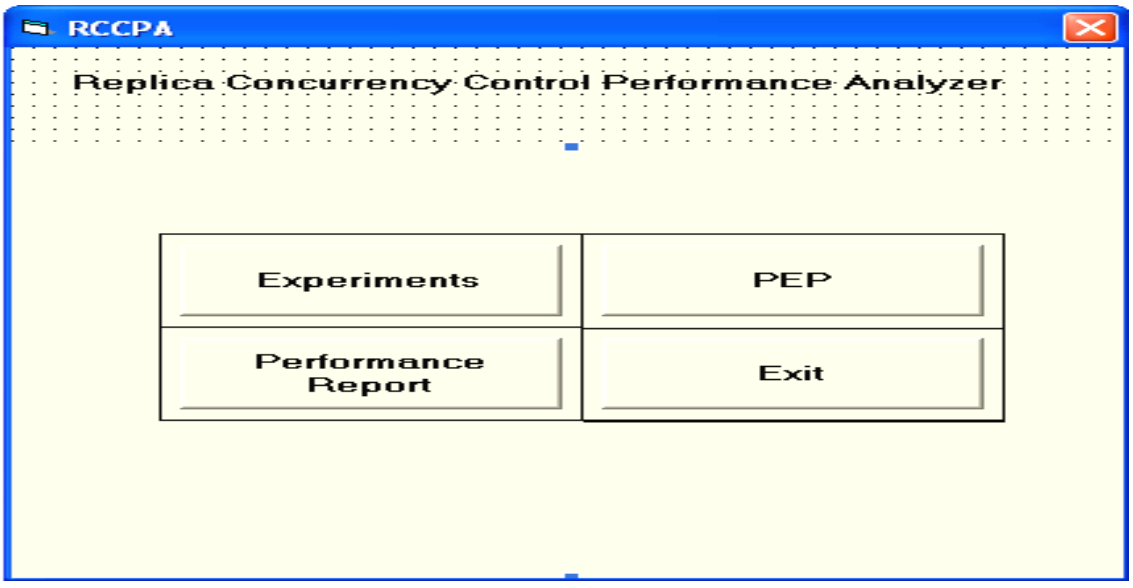

Figure 5.1 Main Screen of RCCPA

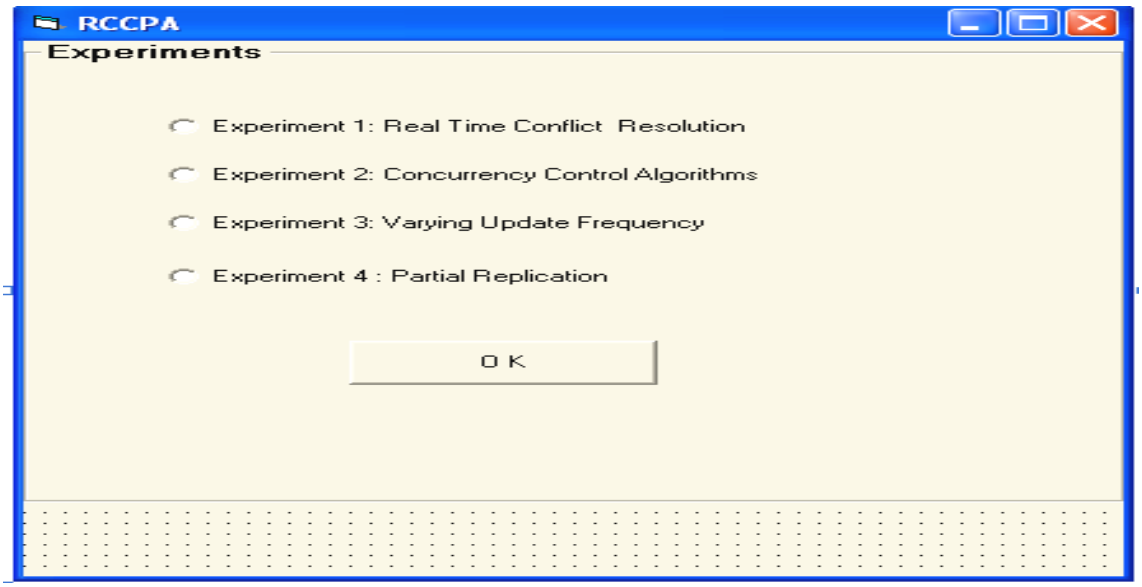

Figure 5.2 Experiment Selection Menus

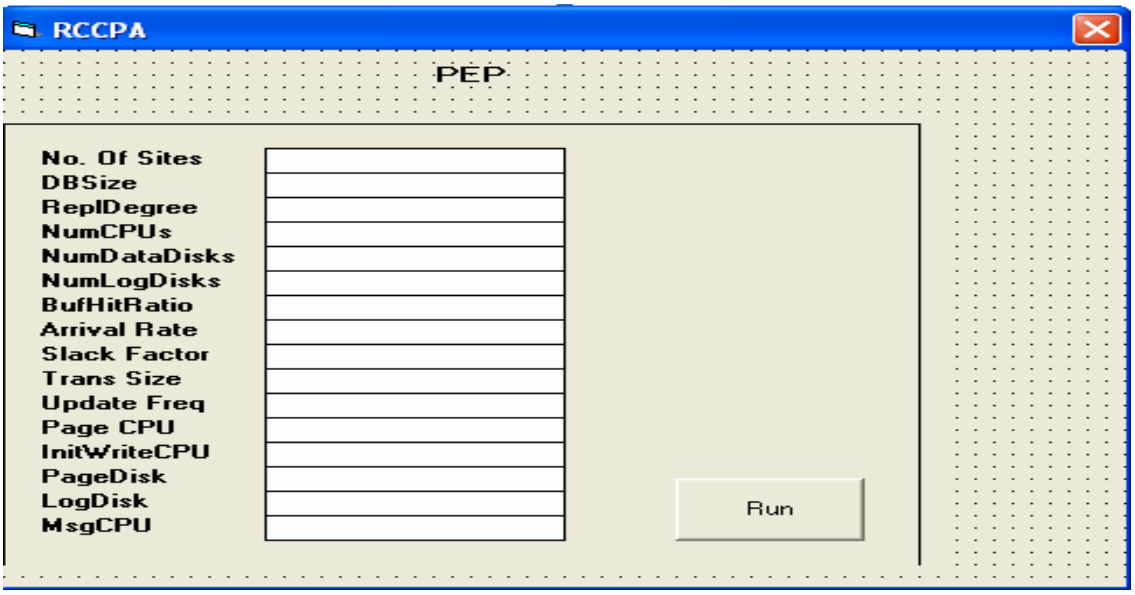

Figure 5.3 Parameter Stetting Component of RCCA

Chapter 6: Experiments and Results

6.1 Overview

This study performs four different experiments to evaluate the performance of replica concurrency control algorithms. First experiment evaluates the performance of the various conflict resolution mechanisms (PA, PB, PI and PA_PB) when integrated with the 2PL and O2PL concurrency control protocols. Experiment 2 evaluates the performance of CC protocols based on the three different techniques: 2PL, O2PL and OCC. Experiment 3 is performed to evaluate the performance of these algorithms under different update frequencies. Experiment 4 is performed to evaluate the performance of these algorithms while varying number of replicas.

The performance metric employed for all experiments is *MissPercent, the percentage of transactions that miss their deadlines.* MissPercent values in the range of 0 to 30 percent are taken to represent system performance under "normal" loads, while MissPercent values in the range of 30 to 100 percent represent system performance under "heavy" loads. Several additional statistics are used to aid in the analysis of the experimental results, including the *abort ratio*, which is the average number of aborts per transaction, the *message ratio*, which is the average number of messages sent per transaction, the *priority inversion ratio (PIR),* which is the average number of priority inversions per transaction, and the *wait ratio,* which is the average number of waits per transaction. Further, the *useful resource utilization* is also measured as the resource utilization made by those transactions that are successfully completed before their deadlines. All the missed deadline percentage graphs in this study shows mean values that have relative half widths about the mean of less than 10% at the 90% confidence interval, with each experiment having been run until at least 10000 transactions are processed by the system. Only statistically significant differences are discussed here.

6.2 Experiment 1: Baseline – Real-Time Conflict Resolution

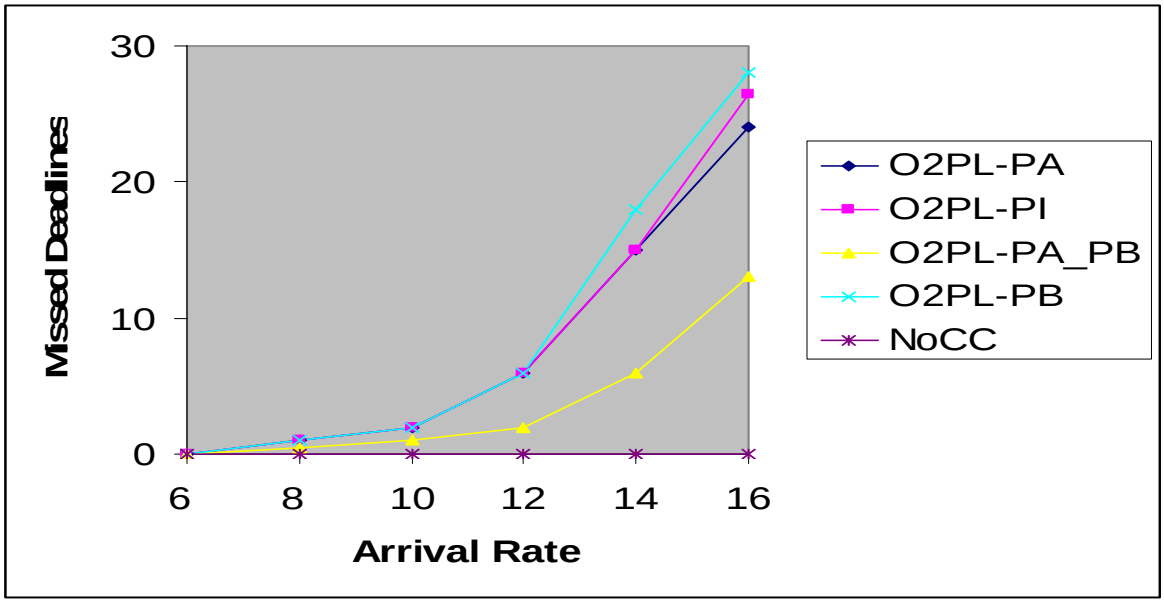

Figure 6.1.1 O2PL- based Algorithms (Normal Load)

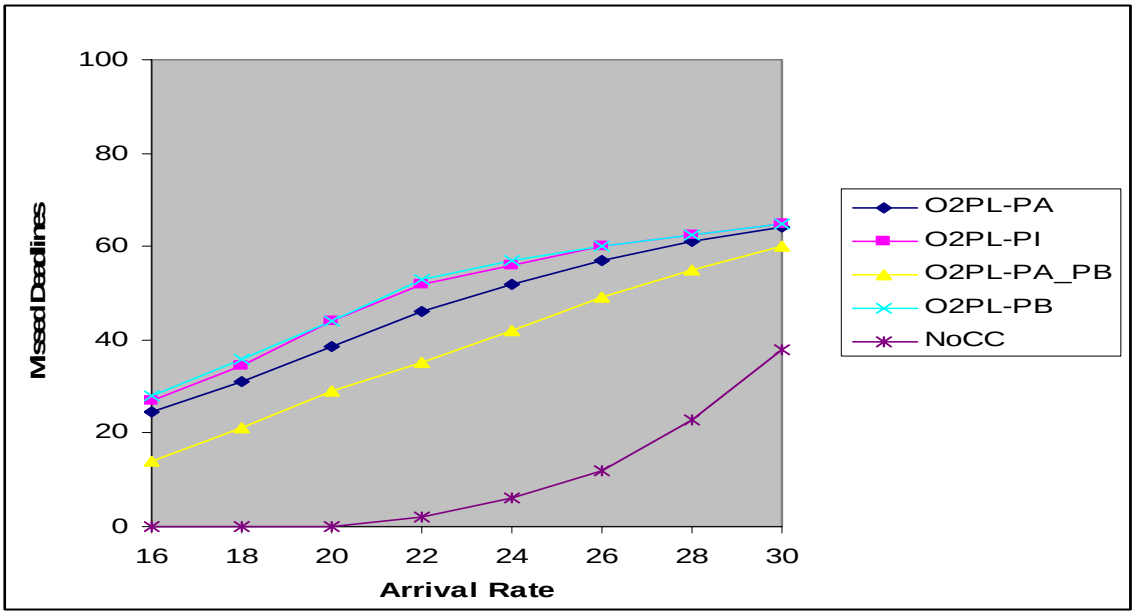

Figure 6.1.2 O2PL- based Algorithms (Heavy Load)

Table 5 presents the setting of the performance evaluation model parameters for our first experiment. With these settings, the database is fully replicated and each transaction executes in a sequential fashion (note, however, that the execution of replica updaters

belonging to the same cohort is always in parallel). The parameter values for CPU, disk and message processing times are similar to those in [16]. While these times have certainly reduced due to technology advances in the interim period, this study also continue to use them here for the following reasons:

1) To enable easy comparison and continuity with the several previous studies that have used similar models and parameter values;
2) The ratios of the settings, which is what really matters in determining performance behavior, have changed a lot less as compared to the decrease in absolute values;
3) This study's objective is to evaluate the relative performance characteristics of the protocols, not their absolute levels. Here the database size represents only the "hot spots", that is, the heavily accessed data of practical applications, and not the entire database.

Here, objective of this experiment was to investigate the performance of the various conflict resolution mechanisms (PA, PI and PA_PB) when integrated with the 2PL and O2PL concurrency control protocols. Since the qualitative performance of the conflict resolution mechanisms was found to be similar for 2PL and O2PL, for ease of exposition and graph clarity, the O2PL-based performance results are only presented here.

For this experiment, Figures 6.1.1 and 6.1.2 present the missed deadline percentages of transactions for the O2PL-PB, O2PL-PA, O2PL-PI, and O2PL-PA_PB protocols under normal loads and heavy loads, respectively. To help isolate the performance degradation arising out of concurrency control, the performance of NoCC (is a protocol which processes read and write requests like O2PL, but ignores any data conflicts that arise in this process and instead grants all data requests immediately) is also presented. It is important to note that NoCC is only used as an artificial baseline in our experiments.

Focusing the attention first on O2PL-PA, it is observed that O2PL-PA and O2PL-PB have similar performance at arrival rates lower than 14 transactions per second, but O2PL-PA outperforms O2PL-PB under heavier loads. This is because O2PL-PA ensures

that urgent transactions with tight deadlines can proceed quickly since they are not made to wait for transactions with later deadlines in the event of data conflicts. From collected statistics, it is found that O2PL-PA greatly reduces the priority inversion ratio, the wait ratio and the wait time as compared to O2PL-PB. The performance of O2PL-PI and O2PL-PB is virtually identical. This is because

(1) a low priority transaction whose priority is increased holds the new priority until it commits, i.e., the priority inversion persists for a long time. Thus, higher priority transactions which are blocked by that transaction may miss their deadlines. In contrast, normal priority inheritance in real-time systems only involves critical sections which are usually short so that priority increase of a task only persists for a short time, i.e., until the low priority task gets out of the critical section. This is the primary reason that priority inheritance works well for real-time tasks accessing critical sections, but it fails to improve performance in real-time transaction processing.

(2) it takes considerable time for priority inheritance messages to be propagated to the sibling cohorts (or updaters) on different sites, and

(3) under high loads, high priority transactions are repeatedly *datablocked* by lower priority transactions. As a result, many transactions are assigned the same priority by “transitive inheritance” and priority inheritance essentially degenerates to “no priority”, i.e., to basic O2PL, defeating the original intention. This is confirmed by the similar priority inversion ratio (PIR), wait ratio and wait time statistics of O2PL-PI and O2PL-PB collected in the experiments. Hence, it is concluded that priority inheritance does not help to improve performance in distributed environment.

6.3 Experiment 2: Baseline - Concurrency Control Algorithms

The goal of this experiment was to investigate the performance of CC protocols based on the three different techniques: 2PL, O2PL and OCC. For this experiment, the parameter settings are the same as those used for Experiment 1. The missed deadline percentage of

transactions is presented in Figures 6.2.1 and 6.2.2 for the normal load and heavy load regions, respectively.

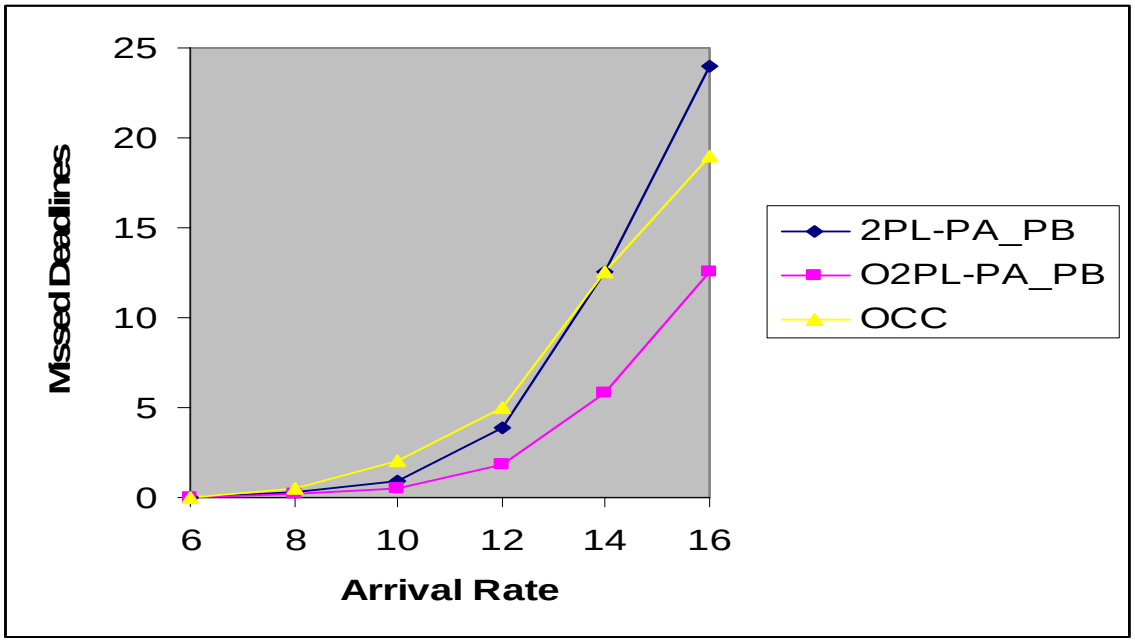

Figure 6.2.1 2PL, O2PL, and OCC Algorithms (Normal Load)

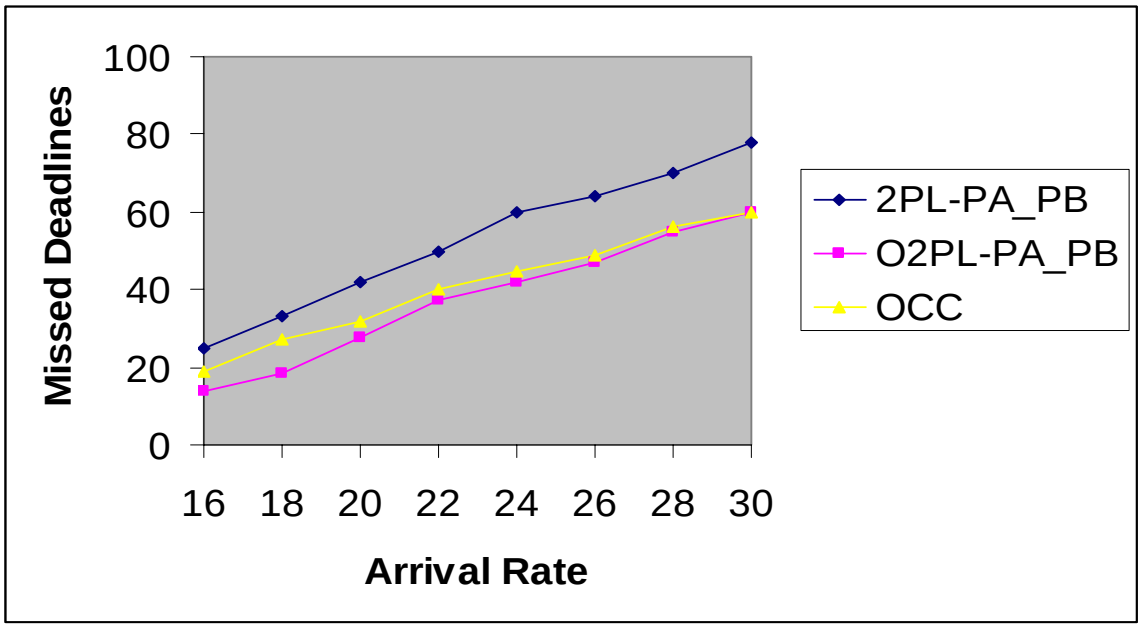

Figure 6.2.2 2PL, O2PL, and OCC Algorithms (Heavy Load)

Focusing the attention on the locking-based schemes, it is observed that O2PL-PA_PB outperforms 2PL-PA PB in both normal and heavy workload ranges. For example, O2PL-PA_PB outperforms 2PL-PA PB by about 12% (absolute) at an arrival rate of 14 transactions/second. This can be explained as follows: First, 2PL results in much higher message overhead for each transaction, as is clearly indicated by the message ratio statistic collected in the experiments. The higher message overhead results in higher CPU utilization, thus aggravating CPU contention. Second, 2PL-PA PB detects data conflicts earlier than O2PL-PA_PB. However, data conflicts cause transaction blocks or aborts. 2PL-PA PB results in more number of waits per transaction and longer wait time per wait instance. Thus 2PL-PA PB results in more transaction blocks and longer blocking times than O2PL-PA_PB. On the other hand, O2PL-PA_PB has less transaction blocks. In other words, unlike in 2PL-PA PB, a cohort with O2PL cannot be blocked or aborted by data conflicts with cohorts on other sites before one of them reaches the commit phase.

Thus, with O2PL-PA_PB, transactions can proceed faster. On the other hand, O2PL-PA_PB improves performance by detecting global CC conflicts late in the transaction execution thereby reducing wasted transaction aborts.

Turning the attention to the OCC protocol, it is observed that OCC is slightly worse than 2PL-PA_PB and O2PL-PA_PB under arrival rates less than 14 transactions/second. This is due to the fact that OCC has a higher CC abort ratio than 2PL-PA_PB and O2PL-PA_PB under those loads. With higher loads, OCC outperforms 2PL-PA_PB because OCC has less number of wasteful aborts, less number of waits and shorter blocking time of a transaction than 2PL-PA_PB. It may be considered surprising that O2PL-PA_PB has the best performance over a wide workload range, improving slightly even over OCC. It is observed that O2PL-PA_PB has higher useful CPU and disk utilization, even though its overall CPU and disk utilization is lower than OCC. This clearly indicates that OCC wastes more resources than O2PL-PA_PB does. It implies that the average progress made by transactions before they were aborted due to CC conflicts is larger in OCC than that in O2PL-PA_PB. As observed in the previous studies of centralized RTDB settings [10], the wait control in OCC can actually cause all the conflicting transactions of a validating transaction to be aborted at a later point in time, thereby wasting more resources even if

OCC has slightly less CC abort ratio than O2PL-PA_PB. In contrast, O2PL-PA_PB reduces wasted resources by avoiding transaction aborts after cohorts/updaters reach demarcation points. In summary, although OCC outperforms 2PL-PA_PB, O2PL-PA_PB, the protocol of O2PL augmented with PA_PB, outperforms OCC in the tested workloads.

6.4 Experiment 3: Varying Update Frequency

The next experiment investigates the performance of these algorithms under different update frequencies. For this experiment, Figure 6.3.1 and 6.3.2 present the missed deadline percentage when the update frequencies are low and high for an arrival rate of 14 transactions/second. It should be noted that data is normally replicated in distributed database systems only when the update frequency is not very high. Therefore, the high update frequency results that are presented here are only to aid in understanding the tradeoffs of different protocols. When the update frequency is comparatively low (less than 0.5), we observe that the qualitative behavior of the various algorithms is similar to that of Experiment 1. A difference, however, occurs when the update frequency is high (more than 0.5). We observe in Figure 6.3.2 that the performance of O2PL-PA_PB degrades more drastically with the increase of update frequency. For example, O2PL-PA_PB performs slightly worse than both 2PL-PA_PB and OCC when the update frequency is 1.0. The reason for the degraded performance of O2PL-PA_PB is that with high update frequency, O2PL-PA_PB causes much more aborts due to both data contention in the local site and global update conflicts, as discussed earlier in Section 6.3, and more aborts are wasted under O2PL-PA_PB. In summary, for low to moderate update frequencies, O2PL-PA_PB is the preferred protocol. For high update frequencies, on the other hand, OCC performs better than O2PL-PA_PB.

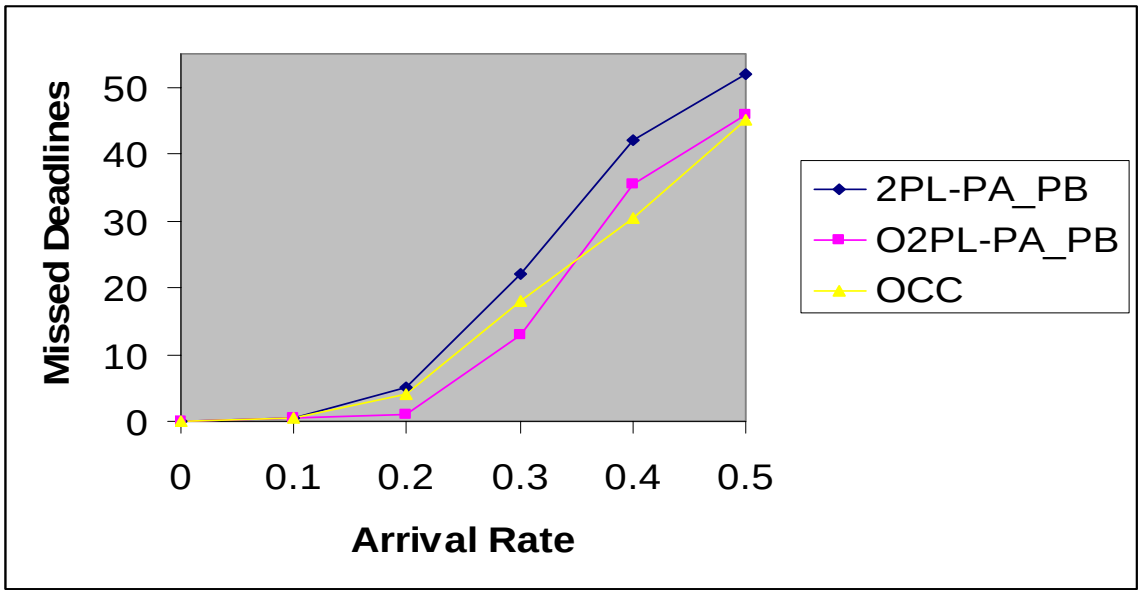

Figure 6.3.1 Varying Update Freq (Low *UpdateFreq*)

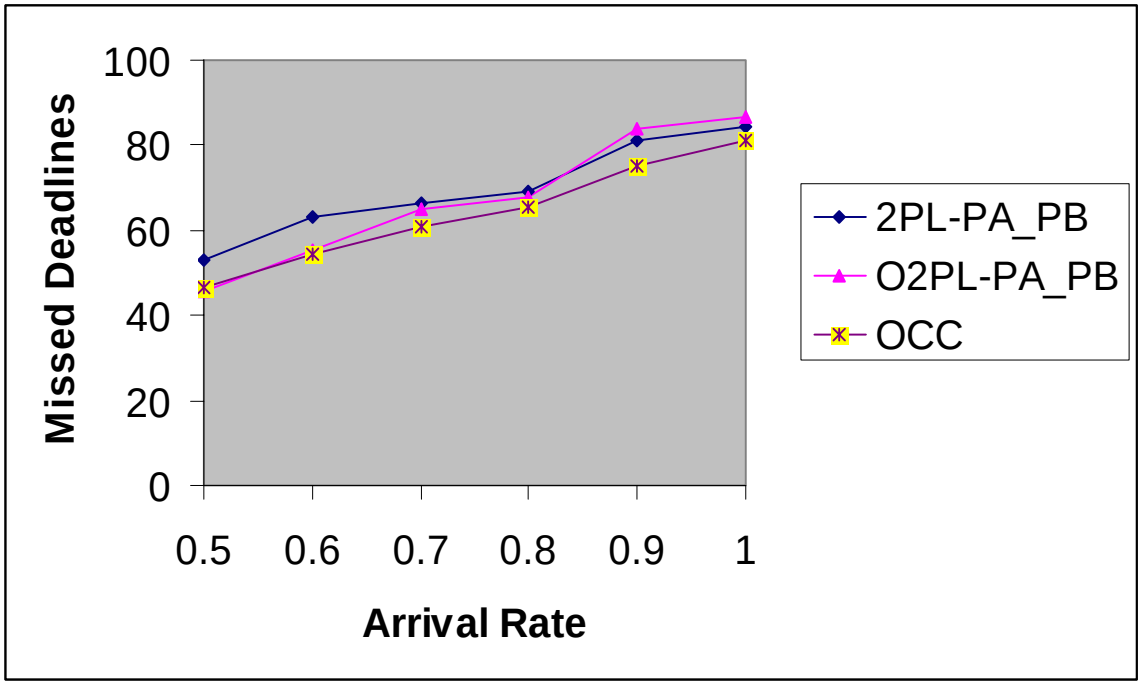

Figure 6.3.2 Varying Update Freq (High *UpdateFreq*)

6.5 Experiment 4: Partial Replication

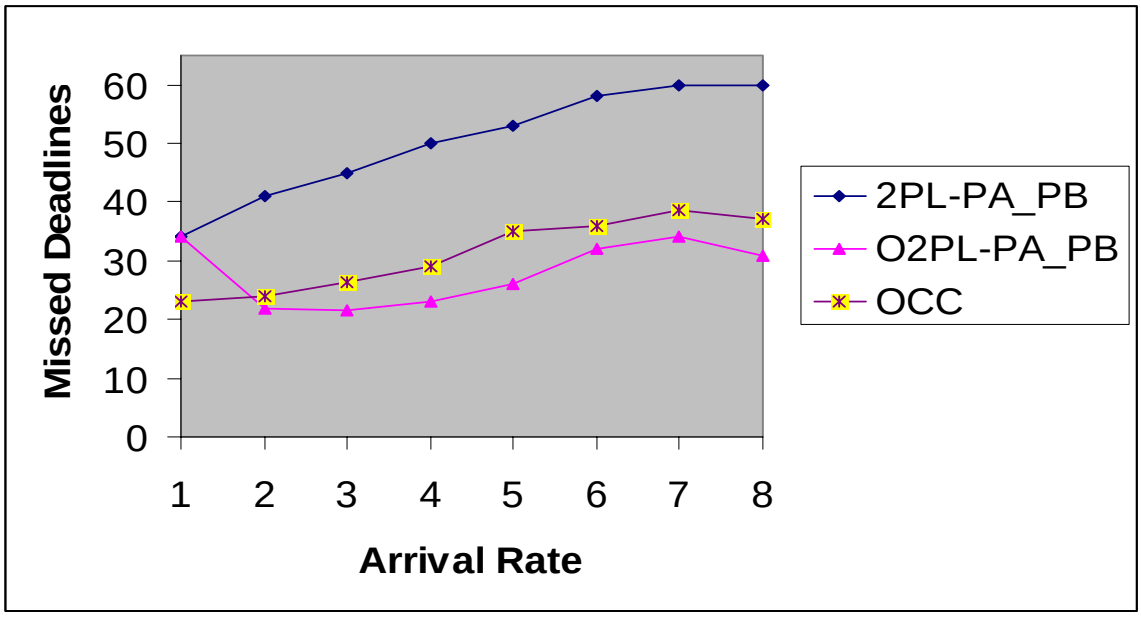

Figure 6.4.1 Partial Replication (*DBSize* = 800, *NumSites* = 8)

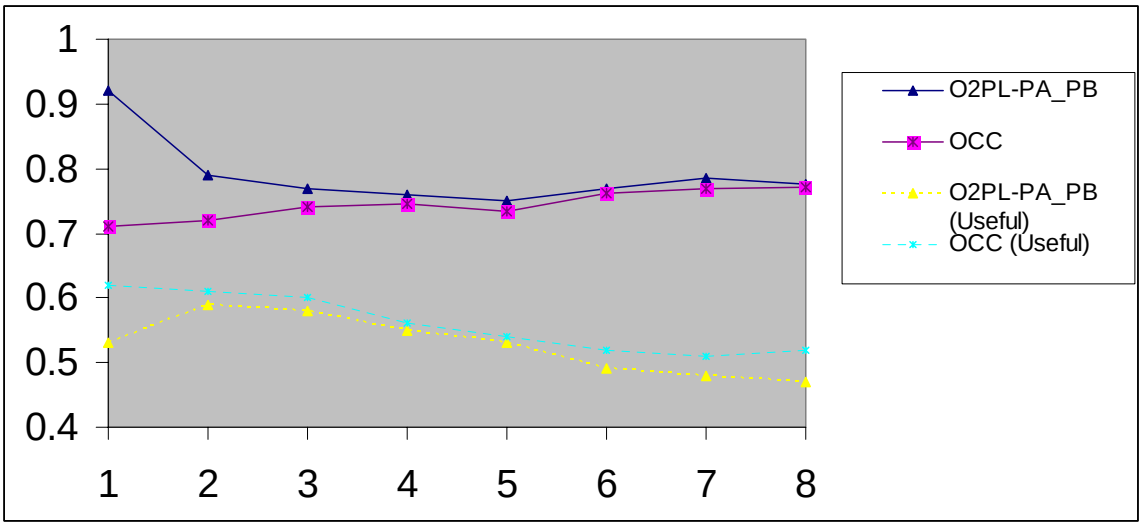

Figure 6.4.2 Partial Replication (Abort Ratio)

For this experiment, the *NumSites* and *DBSize* are fixed at 8 and 800, respectively, while the *NumCPUs* and *NumDataDisks* per site are set at 1 and 2, respectively. These changes were made to provide a system operational region of interest without having to model very high transaction arrival rates. The other parameter settings are the same as those given in Table 5. For this environment, Figure 6.4.1 presents the missed deadline

percentage of transactions when the number of replicas is varied from 1 to 8, i.e., from no replication to full replication, for an arrival rate of 14 transactions/second. In the absence of replication, we observe first that 2PL-PA_PB and O2PL-PA_PB perform identically as expected since O2PL reduces to 2PL in this situation. Further, OCC outperforms all the other algorithms.

As the number of replicas increases, the performance difference between O2PL-PA_PB and 2PL-PA_PB increases. Because of its inherent mechanism for detecting data conflicts, 2PL-PA_ PB suffers much more from data replication than O2PL-PA_PB and OCC do. It is observed that the performance crossover between O2PL-PA_PB and OCC. The reason for this change in their relative performance behavior is explained in the abort curves shown in Figure 6.4.2 (for graph clarity, the abort ratio and useful abort ratio of O2PL-PA_PB and OCC are only shown), where it is seen that the number of aborts of O2PL-PA_PB is significantly reduced while data is replicated. This helps reduce the resource wastage in O2PL-PA_PB. In O2PL, read operations can benefit from local data when data is replicated. However, as data replication level goes up, update operations suffer due to updates to remote data copies. Hence, the performance degrades after a certain replication level. On the other hand, it is observed that the performance of 2PL-PA_PB always degrades as data replication level goes up. This is due to the pessimistic conflict detection mechanism in 2PL since the number of messages sent out for conflict detection increases drastically which in turn increases CPU contention. The similar behavior of OCC and 2PL is also observed in conventional replicated databases [22].

6.6 Summary of Experimental Results

Apart from the experiments described above, a variety of experiments was conducted that cover a range of workloads and system configurations, including “infinite” resources to isolate the impact of data contention, variations in message cost, message propagation delay, slack factor and data access ratio, etc. Table 6 summarizes these results under both tight and loose slack factor: In the table, system parameters, i.e., load, message cost, data access ratio (DAR) and update frequency have been coarsely categorized into low and high, and ’*’ refers to both low and high categories. The terms “poor”, “fair”, “good”,

and “best” are used to describe the relative performance in a given system state and for a given algorithm. Whereas in a particular row, “fair” is better than “poor”, “good” is better than “fair”, and “best” represents the best algorithm in a row, the terms in two different rows are not comparable. The following general observations pertain to Table 6.

1. 2PL based algorithms perform poorly in most cases, especially when the message cost is high. Thus 2PL based algorithms are not the proper choices for high message cost environments.
2. O2PL-PA and O2PL-PA_PB achieve good performance for low to moderate update frequencies but the O2PL approach does not work well at high update frequencies.
3. OCC achieves better performance than all the O2PL based and 2PL-based algorithms over most of the update frequency range.
4. Protocols integrated with only PB or PI (e.g., O2PL-PB, O2PL-PI) always perform poorly. Thus they are not suited to distributed real-time databases. A similar poor performance of these mechanisms has also been observed for centralized real-time databases [10].
5. No single algorithm can always outperform all the others: O2PL-PA_PB performs best for low to moderate update frequencies whereas OCC performs best at high update frequencies. However, since it is expected that most replicated RTDBS applications will belong to the former category, O2PL-PA_PB appears to be the best overall choice for implementation in these systems.

Parameter				**Algorithm’s Performance**								
				2PL				**O2PL**				**OCC**
Load	MsgCost	DAR	UpdateFreq	PB	PI	PA	PA_PB	PB	PI	PA	PA_PB	Wait
Low	Low	High	Low	Poor	Poor	Fair	Good	Poor	Poor	Good	Best	Good
Low	High	High	Low	Poor	Poor	Poor	Poor	Poor	Poor	Good	Best	Good
High	Low	High	Low	Poor	Poor	Poor	Fair	Poor	Poor	Fair	Best	Good
High	High	High	Low	Poor	Poor	Poor	Poor	Fair	Fair	Good	Best	Good
*	*	High	High	Poor	Poor	Poor	Poor	Poor	Poor	Poor	Good	Best
*	*	Low	Low	Fair	Fair	Fair	Fair	Good	Good	Good	Best	Good
*	*	Low	High	Poor	Poor	Poor	Poor	Good	Good	Good	Good	Good

Table 6 Performance of Algorithms

Chapter 7: Conclusions and Further Recommendations

7.1 Conclusions

In this study, the problems of accessing replicated data in distributed real-time databases have been addressed where transactions have firm deadlines, a framework under which many current time-critical applications, especially Web-based ones, operate. In this study, the performance of the 2PL, O2PL, OCC, and O2PL-PA_PB is investigated.

This performance study shows the following:

1. The relative performance characteristics of replica concurrency control algorithms in the real-time environment could be significantly different from their performance in a non-real-time database system. For example, the O2PL algorithm, which is reputed to provide the best overall performance in traditional databases, performs poorly in real-time databases.
2. OCC outperforms 2PL and O2PL based algorithms when these locking based algorithms are integrated with priority blocking, priority abort and priority inheritance protocols.
3. The O2PL-PA_PB protocol provides the best performance in both fully and partially replicated environments for real-time applications with low or moderate update frequencies. For high update frequencies, however, OCC is better. But, given that most of the distributed real time applications that this study is aware of fall into the former category, O2PL-PA_PB appears to be an attractive choice for designers of replicated RTDBS.

7.2 Limitations and Further Recommendations

This study has certain limitations which can be fulfilled by further study. This study specially focused on evaluating the performance of three different replica concurrency control algorithms namely distributed 2PL, O2PL, and OCC when distributed 2PL and O2PL are associated with PA, PB, PI, and PA_PB data conflict resolution techniques and OCC is associated with OPT-WAIT. This study can extend to evaluate the performances of these algorithms when these are associated with PA_PI and PB_PI.

References

[1] Ulusoy, O., "Processing Real–time Transactions in a Replicated Database System", *Distributed and Parallel Databases*, 2, pp. 405-436, 1994.

[2] Philip A. Bernstein, "Transaction Processing", 1999.

[3] Ilyen, "Concurrency Control for Transaction Processing", 2002.

[4] Bharat Bhargava "Concurrency Control in Database Systems", IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 1, January/February 1999.

[5] Philip A. Berstein, Vassos Hadzilacos, Natham Goodman, "Concurrency Control and Recovery in Database systems", Addison Wesley publication, 1987.

[6] Mario Lauria, "Concurrency Control", Ohio State University, Nov 2004.

[7] Berstein P. A., Sihipman D.W., and Wong W.S., "Formal Aspects of Serializability in Database Concurrency Control", IEEE Transactions on Software Engineering, Vol. SE-5, pp. 203-215, May 1979.

[8] Silberschatz, Korth, Sudarshan, "Database System Concepts", pp. 699, Fourth Edition, 2002

[9] M. Casanova, "The Concurrency Control Problem for Database Systems", Ph. D. Thesis, Computer Science Department, Harvard University, 1979.

[10] Gray J., Reuter A. "Transaction processing: Concepts and Techniques", Morgan Kaufmann, 1993.

[11] Barghouti N. S., Kaiser G. E., "Concurrency control in Advanced Database Applications", ACM Computing Survey 23, pp 269- 317, Sept 1991.

[12] Heiko Achuldt, Gustavo Alonso, Hans-Jorg Schek, "Concurrency Control and Recovery in Transactional Process Management", *Proceedings of the ACM Symposium on Principles of Database Systems (PODS'99),* pp 316-326, May/June 1999.

[13] Kumar V., "Performance of Concurrency Control Mechanisms in Centralized Database Systems", Prentice-Hall, 1995.

[14] Robinson J., "Design of Concurrency Controls for Transaction Processing Systems", Ph. D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1982.

[15] Gray, J., “Notes On Database Operating Systems”, in Operating Systems: An Advanced Course, R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[16] Huang, J., Stankovic, J.A., Ramamritham, K., Towsley, D., “Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes”, Proc. of the 17th International Conference on Very Large Data Bases, Barcelona, September, 1991.

[17] Carey, M., and Livny, M., “Conflict Detection Tradeoffs for Replicated Data”, ACM Transactions on Database Systems, Vol. 16, pp. 703-746, 1991.

[18] Haritsa, J. R., Carey, M., and Livny, M., “Data Access Scheduling in Firm Real-Time Database Systems”, the Journal of Real-Time Systems, 4, 203-241, 1992.

[19] Stankovic, J.A., Zhao, W., “On Real-Time Transactions”, ACM Sigmod Record, March 1988.

[20] Liu, C., and Layland, J., “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”, Journal of the ACM, 20(1), 1973.

[21] Agrawal, R., Carey, M., and McVoy, L., “The Performance of Alternative Strategies for Dealing With Deadlocks in Database Management Systems”, IEEE TOSE, Dec 1987.

[22] Ciciani, B., Dias, D. M., Yu, P. S., “Analysis of Replication in Distributed Database Systems”, IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 2, June 1990.