DATABASE CONCURRENCY CONTROL AND RECOVERY IN LOCAL BROADCAST NETWORKS

by

William Kevin Wilkinson

Computer Sciences Technical Report #448

September 1981

DATABASE CONCURRENCY CONTROL AND RECOVERY IN LOCAL BROADCAST NETWORKS

by

WILLIAM KEVIN WILKINSON

A thesis submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1981

DATABASE CONCURRENCY CONTROL AND RECOVERY IN LOCAL BROADCAST NETWORKS

William Kevin Wilkinson

Under the supervision of

Assistant Professor David J. DeWitt

Abstract

Some large business database systems are characterized by a high volume of short transactions (e.g. credit/debit account). In such systems, data retrieval costs are fixed and unavoidable. However, overhead due to concurrency control and recovery protocols may be reduced resulting in higher throughput and shorter response times. This research addresses the problems of concurrency control and recovery (collectively referred to as transaction management) in very large business applications. It is felt that traditional solutions to the problem (i.e. ever-larger centralized machines) will be inadequate as applications grow. An architecture for a database management system distributed over a local broadcast network is proposed. A "passive" concurrency control technique is presented which makes use of the broadcast nature of the communications bus. By eavesdropping for request messages on the broadcast bus, a single concurrency control node can perform conflict analysis for the entire system without explicit lock messages. Two algorithms are presented and shown robust with respect to communication and processor failures: a passive locking algorithm and a passive non-locking algorithm. Simulation results indicate that the passive schemes have very low overhead and perform better than corresponding distributed algorithms. Also, the cost of the recovery protocol necessary to ensure atomic commit at all sites (i.e. the distributed two-phase commit protocol) is shown to be quite high and, in many cases, overshadows the cost of concurrency control.

ACKNOWLEDGEMENTS

There are numerous people who have helped me along the way and I would like thank them at this time. My advisor, Professor David DeWitt, helped me formulate a number of my ideas; provided more than generous support; was a good friend; arranged for an excellent working environment; etc. Professor Ray Bryant provided helpful suggestions in the design and analysis of the simulations; and he developed the SIMPAS preprocessor which saved me much work. Discussions with Professor Bob Cook gave me a better understanding of communication protocols and distributed processing, in general. My colleagues and friends, Haran Boral and Dina Friedland, were always willing to listen to my (sometimes jumbled) ideas and provide constructive criticisms. Haran also provided many lessons in producing a thesis; thanks to him, I did not fall prey to the "me" macros, "refer" or the Thesis Examiner. Finally, I would like to thank my parents for their support and encouragment and Marilyn McGoorty for motivating me to finish.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Problem Environment

Some large business database applications require a high volume of relatively short transactions. Because transactions are short there is a large amount of concurrent activity which must be coordinated. Thus, any system for such a business application should support a high degree of parallelism yet provide fast communication facilities to resolve conflicts. To maintain high throughput, the conflict resolution strategy must incur very little overhead. This thesis proposes an architecture for processing business transactions. We then devise a concurrency control technique for this architecture which has very low overhead. Simulation results show that our concurrency control algorithm performs better than other algorithms and that the architecture is capable of supporting high transaction throughput.

1.1.1. Business Databases

"Business" databases are distinguished from statistical and bibliographic databases in several ways (see Hawtho81a). One major difference is that business transactions access a small number of records one at a time. In contrast, queries in bibliographic and statistical databases require processing of large amounts of data, either for associative search or aggregation. Another difference is that bibliographic and statistical databases tend to be retrieval oriented while business databases can be update intensive. As a consequence, transaction management tasks (i.e. concurrency control and recovery) represent a significantly larger portion of transaction execution time in business systems than in other database applications. For similar reasons, access patterns in statistical and bibliographic databases tend to be more regular since associative search and aggregation is usually performed across an entire relation. Access patterns in business databases are more random.

1.1.2. Proposed Architecture

Traditional architectures for business database management systems have been large centralized mainframes with large amounts of mass storage. However, we feel that this approach is undesirable for the following reasons. First, there is the tendency of applications to outgrow their resources. Databases expand and require additional mass storage units which may involve reconfiguring the database. Eventually, faster processors are needed simply to keep up with increases in the disk bandwidth. A second problem is that a database served by a single machine may be totally inaccessible during machine failures, i.e. reliability is a problem. Alternative hardware support for a business database application might be a database machine architecture (see DeWitt79a, Banerj79a, Ozkara75a). However, these machines are optimized for associative search of large amounts of data (relations in DIRECT and RAP, disk cylinders in DBC). Searching such large amounts of data in order to retrieve a single record would be wasteful and a business transaction might require several such searches. Thus, database machines do not provide a cost-effective solution.

The need for high availability of data coupled with changing resource requirements point to the need for a multi-processor architecture. Local networks of the type exemplified by Ethernet [Metcal76a] are emerging as a flexible, high-speed interconnection medium for processors. Such a network could serve either indirectly as the underlying architecture for a distributed database management system or directly as the communications medium for a loosely coupled database machine.

We propose to distribute the functions of a database management system across processors on a local network. Processors will be dedicated to a single database management task. In particular, "user nodes" will be responsible

for interfacing directly with users and will handle transaction management tasks; "data servers" will access the database, processing read and write requests from the user nodes (Figure 1). We envision user nodes as small personal computers, providing an interface for a single user (e.g., a clerk or reservations agent). Data servers would be more powerful machines, perhaps in the mini or mid-size range. One might also include an index node, a schema node, a sort node, etc. to perform various other tasks. However, we are concerned here only with transaction management issues and their effect on performance. Incorporating other tasks

```
    user             user          ...       user
     |                |                       |
---------------  ---------------        ---------------
| user node  |   | user node  |   ...   | user node  |
---------------  ---------------        ---------------
     |                |                       |
=====================================================
                  <--  ether  -->
=====================================================
          |                              |
    ---------------               ---------------
    | data server |               | data server |
    ---------------               ---------------
          |                              |
       --------                       --------
      /      /                       /      /
     / disk /                       / disk /
    /      /                       /      /
    --------                       --------
```

Figure 1. Sample System Configuration

would make it difficult to directly attribute performance differences to differences in transaction management algorithms (note that this presumes the cost of tasks are independent; we discuss this assumption in Chapter 4). The user nodes and data servers will be described in more detail in Chapter 3.

This architecture solves both problems associated with the centralized machine approach. The system is easily expanded by adding more user nodes or data servers (up to a limit, perhaps determined by the communications bandwidth). And a failure of any single processor or disk would not halt the entire system but would only make part of the database inaccessible. Although it could be argued that the communications cable is a weak link, the cable is so reliable that the only real danger is a faulty interface board which may jam the line with noise.

The usual argument against distributing a business database system is the so-called "Gray" argument (named for Jim Gray, a well-known advocate of this viewpoint). The argument states that distributed processing can only be cost-effective when the amount of work that can be performed by a remote processor outweighs the cost of sending the request. Thus, one gains very little by having a disk node access a single record since the overhead of sending a message is so high relative to the amount of work done by

the disk node. This argument has some merit. However, for very large systems, there seems to be no alternative to distributed processing. If the workload doubles on a centralized system, there may not be a computer available, at any price, with twice the processor speed of the existing machine.

1.2. Primary Interest

This proposed distributed architecture invites many questions such as the assignment of tasks to processors, the interfaces among the tasks, the proper ratio of user nodes to disk servers, etc. However, the primary concern of this research is transaction management (i.e. the concurrency control and recovery tasks) and its impact on throughput. There is no "best" distributed concurrency control algorithm. The choice of an algorithm depends not only on the application but also on the underlying hardware. In this thesis, we have fixed the underlying architecture to one which we feel will become dominant as databases grow and processor speeds stabilize. We then develop a centralized concurrency control technique which takes advantage of the broadcast nature of our architecture. The scheme has very low overhead because, unlike other centralized schemes, transactions do not need prior approval from the concurrency controller before accessing the database. A simulation analysis shows that our scheme performs as well or better than corresponding distributed algorithms.

1.3. Organization of Dissertation

In the next chapter we review some topics in transaction management and survey previous work in concurrency control and recovery. Chapter 3 describes two different centralized concurrency control algorithms for use in our architecture. Chapter 4 describes a simulation model and the experiments done to compare our centralized scheme with distributed algorithms. The final chapter summarizes the results and points to promising areas for future research.

CHAPTER 2

OVERVIEW OF TRANSACTION MANAGEMENT

2.1. Transactions and Consistency

A database is a model of some part of the real world. As such, we are interested in how well the model reflects reality. The state of a database is the union of the values of all records in the database and any internal pointers which link or index records. For every database there is an associated set of rules which can be used to distinguish "consistent" database states from "inconsistent" states. A database state is consistent if it is a good model of reality, i.e. it satisfies the rules. These rules (known as integrity constraints) may impose structural or semantic constraints on the database. For example, a structural constraint may specify the number of entries in the block of a B-tree. Semantic integrity constraints are specific to the application. An airlines reservations system might have a constraint which prohibits someone from making reservations on two flights which overlap in time. A constraint on a banking database might prevent a clerk from entering a negative interest rate. Together these constraints define the permissible database states.

Because a database is constantly being modified, we need a way to guarantee that any change leaves the database in a consistent state. One possibility is to check all integrity constraints after every change to the database. However, while updating the database, one may unavoidably violate an integrity constraint for a short time. For example, a person wanting to change a flight reservation to a different airline may temporarily hold reservations for two flights leaving at the same time. If all constraints were checked after each change to the database then such an action might be disallowed. The source of the problem is that individual changes to the database (i.e. write a record) do not necessarily correspond to changes in the real world. This observation has led to the notion of a database "transaction". A transaction is a series of database actions (e.g., read or write a record) which corresponds to a single change in the real world model. A transaction, then, may be viewed as a function which maps the database from one consistent state to another. Further, it is only necessary to check for integrity violations at the end of a transaction, not during its execution.

In business database systems, however, semantic integrity constraints are not usually represented in the database. Instead, it is assumed that a transaction, given

a consistent database and run alone and uninterrupted will leave the database in a consistent state, i.e. it is assumed that transactions are correctly implemented (including checking for invalid data entered by users). This is a safe assumption for business databases since the database semantics are usually simple and easily checked. Note that for certain types of databases (in particular, databases used to represent large amounts of knowledge about a problem domain), the definition of a consistent state may be complex and change so often that it is no longer safe to assume that transactions maintain consistency. In such systems, it becomes necessary to explicitly check integrity rules after database updates (see Minker78a).

From the previous discussion, it should be apparent that if transactions are run one-at-a-time and the computer never fails, that the database will always be in a consistent state. Unfortunately, computers do fail and restricting database access to a single transaction at a time impairs transaction throughput. As shown in Figure 2, even if individual transactions maintain consistency, database integrity can still be violated by concurrent access by multiple transactions or abnormal termination of a single transaction. This simple database has one integrity constraint: the value of record A must equal the value of

```
Constraint:      A = B

Transactions:      T1:  A,B = 10;      T2:  B,A = 20;

Concurrent Access
                                                A        B
        T1:  write A                            10
        T2:  write B                                     20
        T1:  write B                                     10
        T2:  write A                            20
                                                ----------
                                                20       10

Abnormal Termination
                                                A        B
        T1:  write A                            10
        <- crash ->
                                                ---------
                                                10       ?
```

Figure 2. Violations of Consistency

record B. Consider two transactions which change the values of A and B. The constraint may be violated if transactions are allowed to interleave in arbitrary ways or if a single transaction terminates abnormally. Thus, additional measures must be taken to ensure that consistency is not violated in a "hostile" environment. The mechanism which controls access to the database by multiple users is referred to as the concurrency controller. The mechanism which keeps the database consistent in spite of system failures is the recovery manager.

2.2. Purpose of Concurrency Control

A concurrency control algorithm must guarantee that the database remains in a consistent state when accessed and updated by multiple transactions simultaneously. Recall, that transactions run serially (i.e. one after another) will leave the database consistent. The most common approach to concurrency control, then, is to design an algorithm which interleaves transactions in a way that is "equivalent" to executing the transactions serially. Equivalence may be defined in terms of execution logs, which are just histories of database reads and writes. In a serial log, transactions follow one another. A log of concurrent activity has an equivalent serial log if a transaction sees the same database state in both logs and the final database states are identical (see Figure 3). When a concurrent log has an equivalent serial log the interleaved

```
Transactions:      T1:  A,B = 10;          T2:  A,B = 20;

        Concurrent Log                      Serial Log

         T1: A=10                            T1: A=10
         T2: A=20                            T1: B=10
         T1: B=10                            T2: A=20
         T2: B=20                            T2: B=20
```

Figure 3. Equivalent Logs

transactions are said to be "effectively" executed in the serial order implied by the serial log. In Figure 3, the effective execution order is T1, T2. Figure 2, shown previously, is an example of a log with no equivalent serial execution order. The notion of an equivalent serial log provides a correctness condition for concurrency control algorithms (it is sufficient, but not necessary; when semantic information is available, other conditions may be used [Kung79a] Thus, a concurrency controller may check for serializability by constructing the dependency graph and checking for cycles. Transaction requests which are non-serializable must be prevented by either delaying the request or aborting the transaction.

2.3. Purpose of Recovery

The task of the recovery component of a database management system is to prevent system crashes or aborted transactions from violating database consistency. A basic assumption throughout this section has been that a transaction will maintain consistency if run to completion. The recovery manager must ensure that transactions which are interrupted do not violate consistency. The usual technique for doing this is to restore the database to the state it was in before the transaction began (since, presumably, the database was consistent, then). Any records modified by the transaction are reset to their

previous values. In the abnormal termination example in Figure 2, record A would be restored to its pre-T1 value. Typically, the recovery manager maintains a log of all database changes on stable storage (i.e. storage that survives a system crash, e.g. tape or disk). Restoring the database involves reading the log to discover the database state prior to the transaction.

The recovery manager is also responsible for reliably updating the database. For example, performance or operating system considerations may dictate that database updates be buffered in primary memory until the mass storage devices are free. Thus, a transaction may terminate before its updates have actually been recorded on the disk. A crash at this point would lose the updates since primary memory is not recoverable after a crash. During crash recovery, the recovery manager is called to "complete" any transaction whose updates may have been lost in the crash. This may also be done by reading the log to discover what changes should have been made to the database.

Under the aegis of a recovery manager, transactions are said to be "atomic" because they effectively run to completion or not at all. The inspiration for the term is the notion that a transaction should appear to execute instantaneously, as a single logical change to the database. Although, a debit from a banking account may involve updates to several records in the database, it should appear to the user as a single action which either completes or never happened. The recovery manager implements the "atomic property" of transactions by keeping track of all database changes and undoing or redoing updates as a result of transaction abort or a system crash.

Transaction management, then, consists of two tasks: concurrency control and recovery. A transaction may be viewed as a unit of consistency. The task of the concurrency controller is to interleave requests from database transactions in a manner equivalent to some serial execution of those transactions; the recovery manager implements the atomic property of transactions by restoring the database to a consistent state after transaction abort or a system crash. Thus, the combined effect is that of running transactions serially and instantaneously. The next two sections provide a brief summary of some existing solutions to concurrency control and recovery found in the literature.

2.4. Taxonomy of Concurrency Control Techniques

The previous section introduced the problem of concurrency control and the notion of serializability as a correctness condition. An important result was that a log of transaction requests is serializable if the dependency

graph is acyclic. Unfortunately, this result provides little insight into how to actually construct a concurrency control mechanism. But, the problem has not been overlooked in the literature. In fact, there is a large number of published algorithms, and several surveys have been appeared [Hsiao81a, Bernst81a]. In this section, we identify the basic approaches which we feel form the basis of most concurrency control algorithms. Note that one reason for the diversity of algorithms is the diversity of assumptions which are made. Some common assumptions are as follows. The database is distributed across several processing sites but the amount of data replication varies. The processing sites not only manage a portion of the database but also manage transactions (unlike our system where these functions are performed by separate processors, i.e. user nodes and data servers). Data requests are handled by the local processor whenever possible and the number of non-local transactions is usually varied. Different assumptions are made regarding the processor inter-connection network but it is common to find that messages are nearly as time consuming as I/O requests.

2.4.1. Locking

The earliest and most common concurrency control technique is locking. The method probably evolved from operating system problems where scarce resources (e.g. tape drives) are shared by a large user community. The idea is that a resource must be reserved before it is used and released after it is used. A requestor must wait if the resource is already in use. In a database environment, resources may be records, disk pages, files or even the entire database. And a distinction is made between read access and write access. Generally, concurrent read access is allowed but not concurrent write access, i.e. only one transaction at a time may hold a write lock on a resource.

It is clear that locking can prevent concurrent transactions from interfering with each other's writes. But, our primary concern is maintaining database consistency. An important result in [Eswara76a] showed that any locking system which requires that transactions acquire all their locks before releasing any locks will maintain database consistency. Another result, in [Gray76a], states that unless write locks are held until transaction termination, a transaction abort may propagate to other transactions. For example, suppose transaction T1 modifies record A and releases its write lock. Then suppose transaction T2 reads the new value of A. If T1 subsequently aborts, transaction T2 must be aborted also since it read a record value which never really existed. Together, these rules imply that transactions must hold all locks until termination. Intuitively, this is reasonable. Locking requires that transac-

tions wait until their resources are available. This forces conflicting transactions to run serially.

Some major questions in locking systems concern responsibility for granting locks and checking deadlock. In a distributed database, the problems are even more complex because the conflict information may also be distributed. For example, suppose each site manages locks for its local data. Presumably, a site can unilaterally resolve conflicts involving local transactions (transactions initiated and run locally). The problem occurs when non-local transactions conflict. If a non-local transaction waits, the site must know (or find out) the pattern of waits at other sites in order to avoid global deadlock (Figure 4). In distributed systems, there are basically two options for granting locks: central authority or distributed authority. Centralized locking implies central-

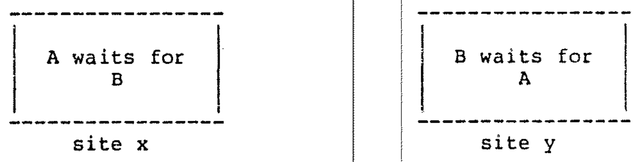

Figure 4. Global Deadlock

ized deadlock detection. However, under distributed locking (i.e. each site grants it own locks), deadlock detection may be centralized or distributed.

If locks are distributed, then we need a way to detect or avoid global deadlock. Algorithms to detect deadlock at a central node are described in [Gray78a, Stoneb78a]. Sites periodically send their parts of the conflict graph to the global deadlock detector which pieces together partial conflict graphs and checks for cycles. A distributed deadlock avoidance algorithm is described in [Lomet80a]. This scheme requires that transactions predeclare any resources that they might access. Another distributed deadlock avoidance scheme (proposed in Rosenk78a), is to attach system-wide unique timestamps to transactions. When conflicts occur between non-local transactions, the younger transaction either waits for the older transaction or it is aborted and restarted (these are actually two different algorithms). Thus, in their scheme conflict resolution is distributed (at the expense of additional transaction restart).

The use of a central lock controller is suggested in [Stoneb76a]. In this scheme (often called primary site locking), all lock requests are first directed to the lock controller which must approve the request before access is permitted. This greatly simplifies conflict analysis since

the information is already at one site. However, the disadvantage is the vulnerability of the scheme when the lock site crashes. Also, there is a potential bottleneck at the central site under high load conditions. Robust central locking algorithms are proposed in [Menasc80a] which uses a reliable communications protocol to replicate conflict information and nominate a new controller in case the central site fails.

When the database is replicated, additional measures must be taken to ensure that all copies of a record converge to the same value. One possibility is to designate one copy as the primary copy (cf. primary site above). All lock requests for that record are sent to the site holding the primary copy. In this way, database locking is distributed but control over a single record is centralized. As in the primary site algorithm, reliability problems result if a primary copy site crashes. A scheme to nominate a new primary copy is presented in [Alsber76a]. Global deadlock detection techniques must still be used as above. Another technique for replicated data is the voting protocol described in [Thomas79a]. At transaction termination, all sites participating in the transaction vote to accept or refuse the pending updates of the transaction. The transaction is committed if a majority of the sites vote to accept the updates. The algorithm is robust with respect to site failures since only a majority of the sites need be up in order to commit the transaction.

2.4.2. Timestamps

Concurrency control methods based on timestamps differ fundamentally from locking approaches. In locking, the ordering of transactions in a serialization is determined while the transactions are executing (based on the interleaving of requests). In timestamp algorithms, the transactions are ordered a priori, based on system-wide unique timestamps. The idea behind timestamp schemes is that transaction requests are always handled in timestamp order. For example, if three transactions want to update the same record, the updates will be applied in timestamp order (Figure 5). Conflicts among those same transactions at other sites will be resolved using the same timestamps.

Request	Tran. Timestamp	Action
T1: write X	831	T1 Delayed
.		
T2: write X	829	T2 Done Immediately
.		
.		
T3: write X	830	T3 Done Immediately T1 Write Applied

Figure 5. Timestamped-Based Concurrency Control

Inconsistencies arising from interleaved requests never occur since requests are always processed in the same order. In theory, then, timestamp based concurrency control can be done without inter-site synchronization messages. In a distributed system, however, timestamp algorithms require as many or more inter-site messages as locking algorithms. For instance, in the previous example, the processing site for record X had no idea how many outstanding updates existed for X with earlier timestamps. It does not know how long to wait for conflicting updates, either. The resolution of this problem is what distinguishes many of the timestamp-based algorithms.

The brute force solution is to poll other sites when a request is received to determine if conflicting transactions exist. If the number of sites is large and requests are fairly frequent this approach could be slow. This problem was attacked in a novel way in [Bernst78a]. In their approach, transactions are partitioned into classes based on their readsets and writesets (thus, resources are pre-declared) and require that a class be run at only one site. By pre-analyzing transaction classes it is possible to pre-determine which transaction classes will conflict. Thus, a processing site need only wait for conflicting requests from conflicting transaction classes, which, presumably is a smaller set than the set of all transactions.

Another technique for determining "out-of-date" requests is to tag records with the timestamp of the last transaction which updated it. Then, when a request is made, the timestamp of the requesting transaction is compared with the timestamp of the record. The request is rejected if the timestamp of the requesting transaction is smaller than the record's timestamp (see Bernst78a, Reed78a, Thomas79a).

If timestamps are not associated with records the processing sites still need to know when requests are "out-of-order". Note that the only purpose of timestamps is to impose an ordering on transactions. Other ordering scheme may be used, as well. Thus, it is possible to allocate "timestamps" logically (i.e. sequentially, not using physical time) [LeLann78a, Kaneko79a]. In these schemes, when an update is received, the processing sites know precisely how many transactions have smaller timestamps. The update may be performed when the site confirms that there are no conflicts with lower numbered transactions. This may be done by polling or by having transactions send their updates to all processing sites.

One issue in timestamp systems is responsibility for dispensing timestamps to transactions. By appending site

identifiers to local clocks, sites can generate unique timestamps themselves [Lampor78a, Thomas79a]. Since access priority is determined by timestamps, a site with a slower clock could inadvertently increase its priority. Algorithms are given in [Lampor78a, Belfor79a] which keep clocks at different sites relatively synchronized.

2.4.3. Non-Locking

A problem with the above two approaches is that there is an inherent delay in processing requests. In locking algorithms, the delay is caused by waiting for resources to become available. In timestamp algorithms, processing sites delay requests until conflicting requests with smaller timestamps have been completed. Because business transactions are small, the probability of conflict is relatively lower than in other applications where transactions scan a large portion of the database. Using a simulation model of database locking in a business application, it has been observed that transaction waits are rare and essentially all deadlock cycles are of length two [Gray81a]. Thus, locking and timestamp approaches are too "pessimistic" in that they lead to unnecessary waiting. This observation has led to "optimistic" approaches in which transactions make accesses without delay and only check for serializability as part of termination processing. The following example illustrates the idea (see

Figure 6).

In this scenario, transaction T0 updates record A which has already been read by transaction T1. In addition, T0 will terminate after writing A. In a locking system, transaction T0 must wait until A is released by transaction T1. In a timestamp system, T0 must wait for all reads from transactions with smaller timestamps. If the read by T1 had already been processed and T1 had a larger timestamp, the write would be rejected as out-of-order. Thus, neither a locking nor a timestamp concurrency control algorithm would allow the requests to be processed in the order shown here. In a non-locking system, we permit T0 to access A without delay. However, to guarantee serializability, we must prevent a cycle in the dependency graph. This sequence of requests (i.e. read A, write A) results in

```
                  time  --->

     T0:  .   .   .  write A   terminate
     T1:    read A       .     .     .     .     .

Serialization:  T1 -> T0

Restrictions:  T0 cannot write A
```

Figure 6. Non-Locking Concurrency Control

a dependency between T0 and T1 and imposes the constraint that T1 must precede T0 in any serialization; this is denoted as:

T1 -> T0

Note that the opposite serial order would imply that T1 read a different value for record A (perhaps the value written by T0). Given this initial serialization constraint, the non-locking algorithm must prohibit any action which results in a T0 -> T1 dependency. For example, if T0 updated record B and T1 later attempted to read that value there would be a cyclic dependency between the two transactions. When a transaction terminates, the concurrency controller examines the execution log to find the global lependencies and to determine if the transaction is serializable.

Thus, the essence of non-locking (or "optimistic") :oncurrency control is that transactions run unhindered ıntil termination. Termination processing, then, consists ›f a <u>validation phase</u> (when serializability is checked) and ι <u>commit phase</u>. The way in which serializability is :hecked is what distinguishes the various algorithms. An lgorithm in [Kung81a] uses transaction timestamps to esolve conflicts. When a transaction begins the system otes the latest timestamp allocated, but does not assign he transaction a timestamp until the validation phase. At that time, the system only needs to check for conflicts with transactions with timestamps between the starting timestamp and the validation timestamp. Transactions outside this range could never conflict with the committing transaction.

Badal [Badal79a] proposes using an access log to keep track of the sequence of reads and writes for each data record. The log would record the transaction identifier, the record accessed and any fields modified as well as the total execution history of the transaction up to the access. The log serves a similar purpose as do timestamps in Kung's algorithm. When a record is requested, the access log is consulted to discover which transactions possibly conflict. The concurrency controller must then must then examine the read and write sets of other transactions in the access log to determine if synchronization is required.

An approach proposed for use in System-D [Eswara80a] is to impose restrictions on what a transaction may access when a conflict occurs. In a locking system, a read may be blocked if there were an active writer. System-D would allow the read but restrict the reader from attempting to update the record later since this would be a non-serializable request.

Summary of Concurrency Control

This section briefly discussed three approaches to concurrency control and various implementations. The goal of each approach is to maintain database consistency by interleaving transaction requests such that the result is equivalent to running the transactions serially. Locking approaches do this by mutually excluding conflicting transactions. Timestamp schemes pre-order the transactions and process requests in that order. Non-locking approaches allow arbitrary inter-leaving of requests and validate serializability at transaction termination time. The amount of concurrent activity permitted by each approach reflects the degree of optimism about the probability of conflicts. In locking approaches, no concurrent update activity is allowed. Timestamp algorithms allow concurrent updates so long as the requests occur in the correct order. Non-locking approaches allow the most concurrency and only check for non-serializability at the end of execution.

2.5. Concurrency Control Performance

A major goal of this research is to design and evaluate concurrency control techniques which permit high concurrency and do not degrade performance. In this section, we discuss factors that affect the performance of a concurrency control algorithm and ways in which an algorithm may affect transaction throughput. We also survey some previous work in concurrency control performance analysis.

2.5.1. Performance Factors and Measures

One reason for the many and varied concurrency control algorithms is that each author works under a different set of assumptions. In fact, there is no "best" concurrency control method. The choice depends on the application and the environment under which the application will run. These factors are not all independent and their relative importance varies for each system.

Application Parameters

One application parameter mentioned earlier is the type of application, i.e. business, statistical or bibliographic. The application type determines two subsidiary parameters: transaction size and frequency of updates. Recall that business transactions execute for a fairly short time while transactions in the other applications usually require some associative search. The probability that a statistical query will update the database is very low while updates in business systems are much more frequent. Another important distinction is whether transactions preclaim resources or request resources on-the-fly. For example, deadlock prevention is simple when all resources are requested at once. In some applications,

however, the necessary records are determined during transaction execution so it is not possible to predeclare resources. Another factor affecting the choice of an algorithm is whether or not it is acceptable to undo transactions. In a banking database, for example, it may not be possible to retrieve cash, once it is dispensed from an automatic teller machine. An important application parameter is the expected degree of interference among transactions. An optimistic concurrency control algorithm may perform poorly in a highly contentious environment. Another assumption made in some algorithms is that resources are read before written. In particular, some timestamp algorithms require that a transaction compare the timestamp of a record before and after execution to see if the record has been altered. In a distributed system, the designers of an algorithm must take into consideration the amount of database replication and the degree of locality. If the database is to be partially or fully replicated, the method must include features for keeping all copies consistent. The degree of locality reflects the amount of transaction processing done at the originating site of the transaction. One would expect less message traffic in a system with a high degree of locality than in a system where few requests are handled at the local site.

System Parameters

The underlying operating system and network manager can affect the performance of a concurrency control algorithm in drastic ways. One concern is the resource granularity of the operating system. For example, if the system prohibits concurrent writing on a file, it will be extremely difficult to implement page level locking. Another consideration is the network topology. Algorithms for a fully replicated database might perform better under a completely connected, homogeneous network than under an point-to-point connected, heterogeneous network. A related issue is the message delay and communications bandwidth. If messages are expensive (either due to limited bandwidth or excessive formatting delays) algorithms with low interprocessor communication should do well. On the other hand, if there is ample communications bandwidth, the emphasis should be on increased parallelism. Another question affecting the design of an algorithm is whether or not the system provides guaranteed message delivery, (i.e. does the system provide datagrams or virtual circuit service) although this is more a reliability concern. Finally, some algorithms are better able to take advantage of a broadcast facility than others, and this should be taken into consideration.

Performance Measures

Concurrency control algorithms have been compared on the basis of a number of measures. A common measure in simulations is transaction response time (or equivalently, throughput), i.e. the average time required for a transaction to complete. Some comparisons have been made on the basis of the number of messages required for conflict analysis. The assumption here is that the cost of interprocess communication is so high it will dominate the cost of concurrency control.

In [Ries79a] comparisons are made on the basis of disk utilization at the processing sites. The idea is that increased concurrency control overhead will manifest itself as a decrease in the number of data requests processed. The notions of "promptness" and "coherence" are defined in [Gelenb78a] for replicated databases. They formalize the problem of getting all data copies to converge to the same value. Algorithms are compared on the basis of their promptness in converging and their relative agreement among different copies (coherence). Evaluations of some optimistic algorithms have considered the probability that a transaction's updates will be accepted [Menasc80b, Kung81a].

Finally, a proposal has been made to evaluate concurrency control algorithms on the basis of the average amount of concurrency permitted by an algorithm [Badal80a]. Intuitively, this is the most satisfactory measure since the object of concurrency control is to increase the number of concurrently executed transactions. However, we feel it is unreasonable to isolate the cost of concurrency control from the cost of other components of a database system. For example, a concurrency control algorithm may have very low execution time overhead but place such great demands on primary memory that very little space remains for user programs. Thus, we feel the best performance measure for a concurrency control algorithm is simply transaction throughput. Such a measure may be obtained from a real system or from a detailed simulation model.

2.5.2. Previous Performance Studies

The previous section mentioned some factors which affect the performance of a distributed concurrency control algorithm. The parameter space is quite large and no attempt has been made to exhaustively search it. However, quantitative and qualitative performance analyses have been made on subsets of the parameters and we review some of the results here.

Ries [Ries79a] compared two centralized concurrency control locking algorithms and two distributed locking algorithms (Wound-Wait [Rosenk78a] and the distributed algorithm in [Stoneb76a]) using a simulation model. The centralized schemes were both primary site models and differed in that one scheme released its locks if blocked. In his analysis, all locks were pre-requested. He varied communications bandwidth, the degree of locality, the transaction size, and locking granularity. Summarizing, he found that under a high degree of locality, the decentralized algorithms worked best. Under a low degree of locality, the centralized schemes worked best. However, it is interesting that the performance differences were not that large. This may have been due to a heavy processor load at each site [Badal80a].

Menasce and Nakanishi [Menasc80b] compared a locking scheme with an optimistic timestamp algorithm using an analytic and simulation model. They found the optimistic (non-blocking) approach far superior to the locking algorithm. However, the database size was small in their analysis (a maximum of 200 items) so transactions were forced to lock a large part of the database.

A centralized locking approach was compared to Thomas' distributed voting algorithm (Thomas79a) by Garcia-Molina [Garcia79a]. He assumed a fully replicated database and preclaiming of locks. He found the centralized algorithm provided uniformly superior response time than the voting algorithm.

A centralized locking, distributed locking and a timestamp based algorithm were compared in [Kaneko79a]. The authors reported somewhat better response times for their timestamp algorithm than the other approaches under a fully interconnected network topology. The differences were not large, however. They also found that the centralized algorithm was the best of the three under a star network topology. Presumably, the algorithm was able to take advantage of the topology by locating the lock controller at the central site.

Badal [Badal80a] attempted to consolidate the previous analyses by qualitatively comparing the interdependence of the performance parameters. He considered three types of algorithms: locking, timestamp, and non-locking. His performance factors were degree of centralization, degree of locality, degree of interference and degree of replication. By reasoning about the effects of such factors he can make qualitative statements about their effect on performance. For example, he states that when data objects are preclaimed, a centralized system should perform relatively better than when objects are requested on-the-fly. The results are intuitively reasonable and unify some

previous work, however the model is not predictive.

2.6. Recovery Techniques

The task of the recovery manager is to restore the database to a consistent state. This may be necessary when a transaction aborts (if the database was modified) or after a system failure. The notion of a system failure encompasses not only processor crashes, but also failures in the communications network and unrecoverable losses of secondary storage. We consider a system failure to be "catastrophic" if it results in a loss of any part of stable (i.e. non-volatile) storage. All other failures are non-catastrophic.

2.6.1. Types of Failures

Transaction Abort

It has already been shown how a partially completed transaction may violate database consistency. Recovery in this context is often referred to as "transaction undo" since the updates of the transaction are undone by restoring the database to its previous state. Note that undo is only necessary if the transaction has actually modified the database. If updates can be deferred until the outcome of the transaction is decided (i.e. commit or abort) transaction undo is greatly simplified. Thus, some authors have suggested placing intermediate results in a transaction workspace until it is decided to commit the transaction [Bernst78a, Kung81a]. This solution is feasible when the size of the workspace is fairly small. If transactions are large then managing the workspace becomes as expensive as modifying the database and undoing updates after aborts.

An underlying assumption is that database updates are not "public" until the transaction commits. Otherwise, another transaction might read a modified record which is subsequently restored to its previous value due to an abort. Thus, the modified value never "existed". If the system is willing to keep track of which transaction have read "temporary" updates, then transactions may apply database updates as soon as they are generated. Such a proposal is made in [Davies73a, Bjork73a].

The most common solution to transaction undo is to maintain a log tape or audit trail of all changes made to the database by the transaction. In case of transaction abort, the log is reread to discover which changes were made [Gray81b]. This will be discussed in more detail below.

Lost Messages

The problem of lost messages actually comes under the domain of reliable inter-process communication, which is beyond the scope of this research. From the viewpoint of the database management system, failure of the communication facility is treated the same as a processor failure. The symptoms are identical; inability to communicate with another process. A point of interest is that it is impossible to devise a protocol which guarantees that a message is received within a fixed amount of time or a fixed number of messages [Yemini79a]. This result has implications in coordinating updates for a transaction at multiple processing sites, as described later.

Processor Crash

For the purposes of this research, a processor crash is presumed to be a software failure or a non-catastrophic hardware failure. Restoring database consistency after such a crash is complicated by the fact that multiple transactions may have been active. And in a distributed system, the problem is even more complex because a transaction may be processed at more than one site. In order to commit the transaction, all sites must agree to accept the updates. Robust protocols are needed to coordinate processing sites so that they all reach the same decision.

Catastrophic Failure

Restoring the database to a consistent state after failure of stable storage (e.g. disk head crash, natural catastrophe) is more difficult and often requires manual intervention. Success of any algorithm depends on the extent of the damage. We do not consider such catastrophic errors in this research. See [Dadam80a] for one approach to the problem.

2.6.2. Log Protocols

Solutions to many recovery problems are facilitated by the use of log files. In this section, we describe the common techniques. Much of this discussion is adapted from [Gray78a]. Each processing site maintains a separate log file which is used to record all database changes at that site. For additional protection, the log may be duplexed and recorded on different media (one disk log, one tape log). It is not strictly necessary that the log file be maintained on magnetic storage. What is required is a non-volatile storage medium of fairly large capacity. However, at this point, the most cost-effective mass storage happens to be magnetic secondary storage devices. Unless stated otherwise, we assume the log file is maintained on disk.

A log record may include information such as a transaction identifier, a record identifier, fields modified and the field values before and after the modification. All log records for a particular transaction are usually linked together with pointers. This makes it easier to locate the log records for a transaction for undo and redo operations. A subtle point is that the log must be written before changes are made to the database. Otherwise, a processor crash between a database modification and writing the log record would lose all information about the change. Thus, it would be impossible to undo the update during recovery. This requirement is known as the Write-Ahead Log protocol (WAL).

Note that use of a log file permits certain performance optimizations. In particular, it is no longer necessary to immediately write to the database when an update is requested. During periods of high utilization, the record may be left in a buffer cache until activity subsides or the buffer is needed. Of course, use of log files actually increases the number of secondary storage accesses because of the number of log records written.

2.6.3. Implementing Atomic Commit

Recall that a transaction must appear to execute atomically, i.e. the transaction must run to completion or not at all. Thus, there must be a point during transaction execution, such that, if a system crash occurs before that point, the transaction will be undone and if a crash occurs after it, the transaction will be completed. This point is called the commit point. Reaching the commit point is different for uniprocessor and distributed systems. When a transaction is ready to commit in a uniprocessor environment, it forces its log records to stable storage (log records may be buffered as well as data records) and writes an additional log record which includes the transaction state (committed). During crash recovery, the log file is read and any transactions which do not have an associated commit log record are undone. Transactions which do have a commit record are redone. Thus, the commit point occurs when the commit record is forced to the log file. A crash before or during the write will lose the transaction. However, a successful write of the commit log record will guarantee that the transaction will eventually complete.

Obviously, a problem with the log-based recovery is that the log file keeps growing. Not only does this require large amounts of mass storage but also slows down crash recovery since the entire log file must be read. To prevent this, database checkpoints are taken which permit the system to discard unneeded portions of the log file. One method of taking checkpoints is described in [Gray81b].

In this scheme, a checkpoint is initiated by first writing a checkpoint log record which includes a list of all active transactions. Then, all log records are forced to stable storage and all buffered writes are migrated to disk. The system may then discard all log records up to the first log record of a currently active transaction since that part of the log is no longer needed for transaction undo or redo.

Implementing atomic commit in a distributed system is more complex because all processing sites must unanimously agree to commit the transaction. In this section, we outline a solution proposed in [Lampso79a]. In their scheme (known as distributed two-phase commit), each processing site maintains its own local log file. Among the sites which process a transaction, there is a distinguished site called the coordinator which is responsible for controlling the other sites, called slaves. When the transaction is ready to commit, the coordinator sends a "prepare-to-commit" message to all slaves (Figure 7). The slaves respond by writing all local log records for that transaction to stable storage and then writing a "ready" log record. The "ready" log record implies that the transaction is ready to be committed at the site but the actual outcome is unknown at this point. After writing the log records, a slave sends an acknowledgment to the coordinator. When the coordinator has received all acknowledg-

```
coordinator              slave1               slave2

send prepare                .                    .
     .                 write logs           write logs
     .                 log "ready"          log "ready"
     .                 send ack             send ack
wait all acks               .                    .
decide com or abt           .                    .
send com/abt                .                    .
     .                 log "com/abt"        log "com/abt"
     .                 send ack             send ack
wait all acks               .                    .
done                        .                    .
```

Figure 7. Distributed Two-Phase Commit

ments, it knows that all slaves are prepared to commit or abort the transaction. If a slave does not respond to the prepare message (i.e. it is crashed or is incommunicado) the transaction is aborted. The coordinator sends a commit or abort message to all slaves which then write a commit or abort message to their local log files and send an acknowledgment. At this point the transaction is recoverable at the slave if it crashes. When all acknowledgments are received by the coordinator, it is safe to assume that all sites agreed on the transaction outcome.

If a slave crashes before the "prepare" message, the transaction is aborted. If a slave crashes when a transaction is "ready" but before it is committed or aborted, the transaction outcome is unknown (or "in-doubt") at that

site. The site must then poll the coordinator to determine the outcome of the transaction. Gray reports that during testing of System-R, some transactions remained in-doubt for weeks [Gray81b].

2.6.4. Nonblocking Commit Protocols

Under some circumstances, it is undesirable to abort the transaction if a slave site has crashed. For example, when updating a record which is replicated at several sites, it may be unnecessary to abort the transaction when a copy site is unavailable (since no work was really lost at that site). One option is to simply wait for the site to recover. However, that could be a long wait. Another option is to allow the transaction to commit in spite of the failed site. The problem is that copies of record may no longer be consistent. The inconsistencies have to be resolved during crash recovery when the failed site is reintegrated in the network. Techniques for doing this are described in [Thomas79a, Montgo79a, Skeen81a].

CHAPTER 3

PASSIVE TRANSACTION MANAGEMENT

In this chapter, we describe the passive concurrency control technique in detail. We present and analyze two algorithms: a passive locking algorithm and a passive non-locking algorithm. We show that these schemes are feasible and robust and discuss how this approach can be utilized in systems other than the one we have presumed here (i.e. non-Ethernet architectures).

3.1. Motivation

Recall that our architecture consists of two types of processors (i.e. user nodes and data servers) connected with an Ethernet-like broadcast communications bus. User nodes provide a user interface and monitor transaction execution. The data servers process read and write requests from the user nodes. In our application, most transactions will be small and updates will be frequent. In addition, the record access pattern will be random and transactions will not preclaim their resources. Finally, we assume that the database is equally partitioned among the data servers, i.e. the database is not replicated. Indeed, since the access pattern is random, there are no performance gains from replicating data (only reliability gains). Our task,

then, is to devise the optimal concurrency control algorithm for this architecture.

Because the conflict information is distributed (at the data servers), our first concern is whether to use a centralized or distributed algorithm. An environment similar to the one we have proposed was considered in [Ries79a]. He found that the decentralized algorithms provided better response time than the centralized algorithms. This was due primarily to a heavy load at the central site (even though resources were preclaimed). A contributing factor was limited communications bandwidth since the centralized scheme required more messages than the distributed schemes. This factor became dominant as bandwidth was severely restricted. Note that his model presumed a high degree of locality (i.e. most database requests processed locally); thus, under a distributed algorithm, many transactions could be executed without any inter-processor communication.

In [Badal80a] it is suggested that a centralized scheme is best for applications characterized by a low degree of locality. The justification is the additional messages required for global conflict analysis in a distributed scheme. But, conversely, it is stated that a distributed algorithm is best for applications where resources are not preclaimed. The relative weights of the factors or the joint effect of the factors simultaneously is not considered. Thus, the evidence that a distributed system is better than a centralized system for our application (or vice versa) is not conclusive either way.

All things being equal, we would prefer a centralized algorithm over a distributed algorithm. Centralized schemes are simpler to understand and easier to implement than distributed schemes. It has been estimated that writing a distributed algorithm is up to twice as complex as writing a centralized algorithm [Garcia79b]. However, all things are not equal. For example, in our application, resources are not preclaimed so a centralized scheme incurs the overhead of a lock request message for every database read and write. Even with infinite bandwidth, this would clearly degrade performance due to operating system overhead involved in processing messages. The central idea of this research is that it is possible to take advantage of the broadcast nature of the communications medium. Whenever a read or write request is sent from a user node to a data server, the request can be overheard by all nodes on the network. Thus, a centralized concurrency control node could simply eavesdrop on the communications bus for read and writes messages and do conflict analysis without explicit lock requests. Database synchronization is done "passively" because no additional inter-processor communication

is required by the user nodes or data servers.

Our passive scheme has all the advantages of a centralized algorithm and few of the disadvantages. Its attraction is the potential for increased throughput due to the low overhead of the scheme. However, a potential problem is that the central controller may become a bottleneck and actually decrease throughput. Also, the central node may fail and leave the system without any concurrency control mechanism. Another problem is that the passive concurrency control node might miss a read or write message which would render its conflict information inconsistent with the system state.

The success of our scheme depends on the existence of a broadcast facility in the communications network. There have been a few proposals for using a broadcast capability to advantage. Algorithms for distributed semaphores are presented in [Banino79a]. Parallel sort and project algorithms using a broadcast channel are given in [Boral80a]. A distributed timestamp concurrency control algorithm is given in [Cheng80a], and optimizations for a broadcast channel are described. However, the issue of reliably broadcasting a message has not been addressed. In fact, unless the communication channel is perfect, there can be no algorithm which guarantees that a broadcast message is received by all recipients simultaneously [Yemini79a].

In [Banino79a] a site detects that it missed a broadcast message by comparing version numbers of control variables at a later time. However, we require immediate detection and retransmission of lost messages because the concurrency control node must have timely conflict information. In Section 3.3, we develop protocols which make the passive scheme robust with respect to lost messages, both broadcast and individual messages. We also describe how our system can recover from a failed concurrency control node. In Chapter 4, we analyze the performance of the passive scheme and show that the central node is unlikely to become a bottleneck.

3.2. Passive Concurrency Control

The passive concurrency control schemes require the addition of a third type of processor to our architecture: a "cc node." The cc node implements the passive concurrency control algorithm. It relieves the user nodes and data servers from most of the burden of conflict analysis and greatly simplifies the read, write and commit protocols. Only one cc node is needed per local network since it is capable of monitoring all message traffic. In this section, we present passive versions of the locking and non-locking concurrency control algorithms. However, we first describe how transaction execution proceeds at the user nodes since it is identical for both schemes. In the

discussion below, we assume a perfect communications channel and that no processors crash. Techniques for making the scheme robust are described in Section 3.3.

3.2.1. Transaction Execution

In this discussion, refer to Figure 8 for an example of transaction execution. It illustrates a transaction which reads data records A and B and updates records B and C, without conflicts. We assume records A and B are maintained on the same data server while record C is on a different data server. Messages sent and received are denoted by right and left arrows, respectively. Transaction execution proceeds through three stages: a read phase, a write phase and a commit phase. During the read phase, the user node obtains a set of records (termed the "readset") by sending read requests to the data servers. Since we assume resources are not preclaimed, the requests are sent individually. Also during this phase, records may be updated or new records written. However, the changes are held in a transaction workspace at the user node and are not immediately sent to the data servers.

At the end of the read phase, a transaction selects a subset of the records in its workspace which it sends to data servers as database updates (termed the "writeset"). This also signals the intention of the transaction to ter-

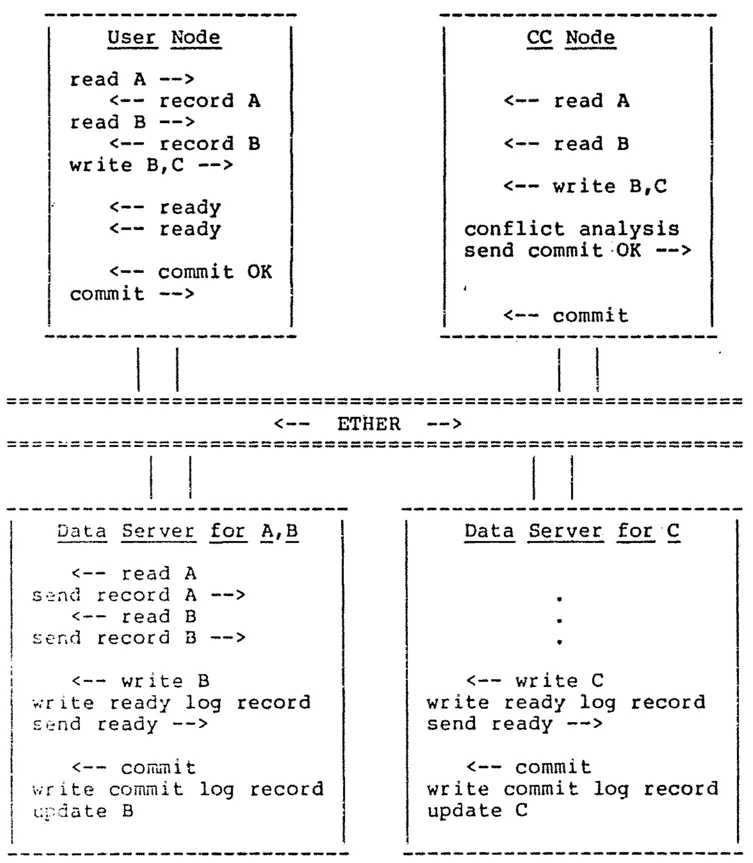

Figure 8. Sample Transaction Execution

minate. Thus, the notion of a separate write phase is

rather fuzzy in this model since that phase coincides with the commit phase. The updates are broadcast to the data servers which select those changes which apply to their portion of the database. The updates are not immediately applied to the database, however. A two-phase commit protocol is used for reliability. Thus, a log record must first be written by the data servers as described in Section 2.6.3. The user node waits for acknowledgments that log records have been written by the data servers before broadcasting a commit or abort message. Only when the data server receives a commit message are the updates applied to the database.

We have omitted a discussion of conflict analysis. When the user node sends its write requests, the cc node overhears the message and marks the transaction as attempting to commit. The cc node then checks the serializability of the requests and when it has resolved all conflicts, it sends a message to the user node which grants the transaction permission to commit. Similarly, if the requests are non-serializable, the cc node informs the transaction that it must abort. Thus, the user node must wait for two conditions before terminating. The recovery protocol requires that it wait for ready log records to be written at the data servers. The concurrency control protocol requires that it wait for the cc node to grant the transaction permission to commit or abort. When these conditions are met, the user node broadcasts a commit or abort for the transaction.

3.2.2. Passive Locking Algorithm

Read and Write Protocols

Our locking requirements are identical to those previously presented in the literature: concurrent read access is permitted but concurrent write access is not. Read requests must be delayed if there is an active transaction which intends to update the record (i.e. has sent a write request but not terminated). A writer must wait until all conflicting readers have released their locks. These rules imply that a data server must keep track of which records are being updated so that it can know when to delay read requests which conflict with pending writes. Unless this is done, the data server may be unaware of an access conflict and would process the read immediately (and the cc node, which is aware of the conflict, is powerless to stop it). In the implementation section below, we describe a simple data structure which the data server uses to keep track of read and write locks on a record. Note that write requests are always delayed until a commit message is received for the transaction. At that point, the data server knows it is safe to apply the updates (since the

commit message is not sent until the cc node has approved the write requests for the transaction).

A transaction does not send a commit or abort message until receiving permission to terminate from the cc node. Thus, the cc node effects synchronization by controlling the order of commit. When the cc node overhears a write (and thus termination) request for a transaction it checks for conflicting locks on the records to be updated. If there is none, the cc node immediately sends the transaction permission to commit. If there is a conflicting reader however, the cc node must delay the writer until it has determined that the request is serializable. Recall that transactions are serializable if the dependency graph is acyclic. When a writer is blocked by a reader, it implies the writer must serialize after the reader (or else the reader would have read the new record value). For such conflicts, the cc node adds an edge from the reader to the writer in its global dependency graph. This signifies that the reader must precede the writer in any serialization. But it is not safe to commit the writer until the complete dependency graph is known. Since the reader may have dependencies caused by its own updates, (and so add edges to the dependency graph) the cc node must wait until the writeset of the reader is known. This procedure repeats until a cycle is detected or all dependencies of all transactions in the graph are known (i.e. the writesets of all transactions are known). Then the writer may commit since the dependency graph has been found to be acyclic.

Notice that the cc node need not wait for conflicting read locks to actually be released before committing a conflicting transaction. It need only wait until the reader declares its writeset. At that point, it can determine if the requests are serializable. For example, see Figure 9. Suppose transactions T1 and T2 conflict over record A and there are no other conflicts in the system. The cc node forces T1 to wait until T2 sends its updates. When this occurs, the cc node determines that there are no additional

```
     T1: write A      T2: read A,B,...,Z, write Z

  T1              T2              cc node

                  read A
write A
                                 T1 must wait on T2
  .               read B
                    .
  .                 .
                  write Z
  .                               T2 ok to commit
                                  T1 ok to commit
                  T2 commit
T1 commit
```

Figure 9. Request Sequence and Order of Commit

conflicts. The dependency graph is known so the cc node grants both T1 and T2 permission to commit. In a strict locking system, transaction T1 would not proceed until the commit message for T2 had been processed. Note that if T2 had also wished to update record A, additional measures would have to be taken to ensure that the updates from T1 and T2 appear in the correct order. In this case, the cc node would not send T1 permission to commit until T2 had sent its termination message.

The above discussion only considered the case of a writer conflicting with a reader. The case of a write request conflicting with another write request is simpler. The cc node must make sure that the updates from both transactions are applied in the same order at all sites. This can be done by granting one transaction permission to commit but delaying the second transaction until the first transaction broadcasts its termination message. In this way, the commit messages for both transactions will be received in the same order at all sites where they conflict.

Implementation

For each active record at its site, a data server maintains an "access log". The access log is essentially a lock manager. It lists the readers and writers of that record and any transactions which are waiting for access (Figure 10). When a read request is received, the requesting transaction is added to the list of readers if there is no writer. Otherwise, it is placed in the wait queue. Similarly, a transaction requesting a write is made the current writer if there are no readers or writers of the record. When a transaction terminates, it is removed from all access logs in which it appears. This may trigger additional activity, as when an update blocks a pending read.

To keep track of global dependencies, the cc node must maintain access logs for each active record (a record is "active" so long as it is in use by a transaction). It must also keep track of the readsets and writesets of transactions. For example, when a read request is blocked by a writer, the cc node must determine if deadlock occurred.

Request Sequence
T1: read X T2: read X T3: write X T4: read X

Access Log for Record X

Readers	Writer	Waiters
T1, T2	-	T3, T4

Figure 10. Request Sequence and Access Log

To do this, it must check the access logs for all records read previously by the reader. Also, when a transaction terminates, the cc node must update the conflict information for all records accessed by that transaction. Thus, it must know the read and write sets for the transaction.

The cc node process a read request as outlined in Figure 11. If the read is blocked by a writer, the request is placed in the wait queue (waitq) for the data record. Also, an edge is added from the writer to the reader in the global serialization graph (i.e. dependency graph). The reader must serialize after the writer because the reader gets the new record value. If a cycle occurs in the graph (i.e. deadlock is detected), the transaction is aborted. If the read request is not blocked, a read lock is set by

```
process read request
    add rec.id to tran.readset
    if rec.writer <> null then
        (* conflict with active writer *)
        add request to rec.waitq
        add edge (rec.writer,tran) to sergraph
        if cycle in sergraph
            abort tran
    else
        (* set read lock *)
        add tran.id to rec.readers
```

Figure 11. Read Request at CC Node: Locking

adding the transaction to the list of readers of that record.

The cc node processing for a write request is illustrated in Figure 12. A request for a write lock may be blocked by a read lock or a write lock. In either case, we add the write request to the wait queue for the data record, and add an edge from the blocking transaction to the requesting transaction. If a cycle occurs in the serialization graph, the transaction is aborted. Note that

```
process write request
    (* request may contain several updates *)
    for each rec in request
        add rec.id to tran.writeset
        if rec.readers <> null
            (* conflict with active reader *)
            add request to rec.waitq
            for each rdr in rec.readers
                add edge (rdr,tran) to sergraph
                if cycle in sergraph
                    abort tran
        else if rec.writer <> null
            (* conflict with active writer *)
            add request to rec.waitq
            add edge (rec.writer,tran) to sergraph
            if cycle in sergraph
                abort tran
        else
            (* no conflicts, set write lock *)
            rec.writer := tran.id
    if all write locks granted
        send tran permission to commit
```

Figure 12. Write Request at CC Node: Locking

write locks and read locks from the same transaction do not conflict with each other (even though the algorithm sketch does not check for this). When a transaction has acquired all of its write locks, it is granted permission to commit. This is equivalent to checking that no edges emanate from the transaction in the serialization graph.

Previously, we discussed an optimization for the case of a write request blocked by a single read request. Recall that the cc node may immediately grant the writer permission to commit upon receiving the readers updates (i.e. it need not wait for the commit message from the reader). That optimization is not shown in the sketch of the write algorithm. It is a simple modification, however, and amounts to releasing the read locks of a transaction once the transaction has been granted permission to commit.

The cc node processing of commit and abort messages is outlined in Figure 13. When the transaction terminates, the cc node simply releases all locks held by the transaction. The transaction is also removed from the serialization graph. Any blocked read or write requests will be reprocessed and will attempt to set locks.

One constraint that the cc node places on the data servers is that data servers process requests in the order received. If the order were somehow changed (due to

```
process commit or abort message
    for each rec in tran.readset
        (* release read lock *)
        remove tran from rec.readers
        for each wrtr in rec.waitq
            (* update sergraph *)
            remove edge (tran,wrtr) from sergraph
            (* try to process pending write requests *)
            for each rqst in rec.waitq
                if rqst.type = write
                    process write request
    for each rec in tran.writeset
        (* release write lock *)
        rec.writer := null
        (* update sergraph and try to process
        ** pending requests *)
        for each rqst in rec.waitq
            case rqst.type
            read:   rdr := rqst.tran.id
                    remove edge (tran,rdr) from sergraph
                    process read request
            write:  wrtr := rqst.tran.id
                    remove edge (tran,wrtr) from sergraph
                    process write request
```

Figure 13. Termination Message at CC Node: Locking

sophisticated scheduling, for example) the access logs would not be consistent with the conflict information at the cc node. Thus, the cc node would be unaware of the exact data dependencies resulting from the interleaved requests.

Correctness

The locking scheme presented above is trivially correct when one considers its equivalence with previous

locking algorithms. It was shown in [Eswara76a] that concurrent-read/single-writer locking schemes maintain consistency when locks are held until termination. Also, Papadimitriou [Papadi79a] showed that a series of interleaved transactions was serializable if and only if the dependency graph is acyclic. The data servers enforce the lock-before-read protocol. The cc node enforces the lock-before-write protocol by preventing transaction commit until the dependency graph is checked for cycles.

3.2.3. Passive Non-Locking Algorithm

The motivation for the non-locking algorithm is the potential for increased throughput as a result of decreased waiting. For example, when a transaction wants to update a record it must wait for all readers of that record to declare their writesets. If those readers must wait in turn, the original writer must wait, too. There is no limit on how long a transaction must wait (i.e. starvation is a problem). The purpose of waiting is to check for cycles in the dependency graph. The passive non-locking algorithm provides a way of preventing cycles in the dependency graph without waiting for the conflicting reader to terminate. Figure 14 (adapted from a previous example in Figure 9), shows how this may be done. Note that in this scheme, transaction T1 does not have to wait for T2 to declare its updates. The cc node prevents a cycle in the

```
     T1: write A      T2: read A,B,...,Z,   write Z

   T1              T2              cc node

                read A
write A
                              T1 ok to commit
                              T2 may not write A
T1 commit
                read B
                  .
                  .
                  .
                write Z
                              T2 ok to commit
                T2 commit
```

Figure 14. Non-Locking Request Sequence and Commit Order

dependency graph by prohibiting T2 from ever updating record A. We refer to these access constraints as restrictions lists. In the locking scheme, the commit order is the same as the serialization order (because writers must wait). In the non-locking scheme, the transactions may commit "out of order" because the cc node prevents cyclic dependencies from developing later. Note that repeatable reads of record A are possible for T2. The old value of A is cached in a workspace at the user node of T2.

Read and Write Protocols

With respect to the user nodes, the read, write, and commit protocols for the passive non-locking algorithm are identical to those in the passive locking scheme. For the data servers, the only difference is that read requests no longer block if there is an active writer. It is assumed that the cc node will detect any non-serializable request sequences and abort the offending transaction. Thus, in this scheme, the data servers have no responsibility for conflict analysis (unlike the passive locking algorithm). Read requests are processed immediately. Write requests are temporarily held until the transaction terminates, when they are applied to the database. Note that the algorithm is truly non-blocking because the cc node can decide to commit or abort the transaction immediately upon receiving the writeset. Rather than being delayed, an "out-of-order" commit will cause access restrictions to be placed on other active transactions (e.g. T1 above).

Implementation

As before, the cc node must keep track of the read and write sets of each active transaction. In addition, it must maintain a restrictions list for each transaction. The restrictions list is a list of records that may not be read or written by the transaction, lest a cyclic dependency occur. When the cc node receives a read request for a record, it process the request as outlined in Figure 15. The "set restrictions" procedure is called when transaction X must serialize before transaction Y (due to some read, write request sequence). Restrictions are recursively

```
set restrictions (x,y)
    (* tran x is serialized before tran y.
    ** update restrictions list for tran x.
    ** restrictions are imposed by tran y are:
    -- x may not read anything written by y
    -- x may not write anything read by x
    -- x may not write anything written by x *)
    add y.writeset to x.noread
    add y.writeset to x.nowrite
    add y.readset to x.nowrite
    (* propagate restrictions to other
    ** tran which serialize before x *)
    for each edge (z,x) in sergraph
        set restrictions (z,y)

process read request
    add rec.id to tran.readset
    add tran.id to rec.readers
    if rec.id is in tran.noread
        (* access violates restrictions list *)
        abort tran
    if rec.writers <> null then
        (* conflict with all active writers *)
        for each wrtr in rec.writers
            (* serialize before each active writer *)
            (* since the old record value is read *)
            add edge (tran,wrtr) to sergraph
            if cycle in sergraph
                abort tran
            set restrictions (tran,wrtr)
```

Figure 15. Read Request at CC Node: Non-Locking

imposed on any transactions which precede X in the serialization as well. If this were not done, cycles involving more than 2 transactions would not be detected (this is described in more detail below).

When the read request is received, the cc node first checks that the read does not violate an access restriction. Then a check is made for any active writers. If there is none, the request is simply noted in the access log and in the readset of the transaction. An active writer, however, implies a dependency and thus a serialization constraint. Specifically, the reader must precede the writer in any serialization. This is because the reader got the old record value (the writer is still in the two-phase commit protocol and has not yet sent a commit message). Although we do not know the status of the writer (it may have been aborted), we must assume that it has been granted permission to commit by the cc node. Thus, we can only prevent a cyclic dependency by placing access restrictions on the reader. A cyclic dependency could occur in two ways: if the reader wrote or read any record updated by the writer or if the reader wrote a record read by the writer. Either action would result in a dependency which would serialize the writer before the reader; a case that must be prevented. Thus, the read restrictions include the write set of the writer and the write restrictions are the read and write sets of the writer.

The cc node processes a write request as outlined in Figure 16. Because the algorithm is non-blocking, the cc node must immediately decide to commit or abort the transaction and send its response. Any conflicting readers must serialize before the writer. Thus, they are res-

```
process write request
    (* request may contain several updates *)
    for each rec in request
        add rec.id to tran.writeset
        add tran.id to rec.writers
        if rec.id is in tran.nowrite
            (* access violates restrictions list *)
            abort tran
        if rec.readers <> null
            (* conflict with active readers *)
            for each rdr in rec.readers
                (* must serialize after readers *)
                (* since readers have old record value *)
                add edge (rdr,tran) to sergraph
                if cycle in sergraph
                    abort tran
                set restrictions (rdr,tran)
        if rec.writer <> null
            (* conflict with active writers *)
            for each wrtr in rec.writers
                (* serialize after existing writers *)
                add edge (wrtr,tran) to sergraph
                if cycle in sergraph
                    abort tran
                set restrictions (wrtr,tran)
    (* if get this far without abort, may safely commit *)
    send tran permission to commit
```

Figure 16. Write Request at CC Node: Non-Locking

tricted as above. A cyclic dependency between the reader and the writer will cause the writer to be aborted. But the cc node is not finished at this point. It must also check for cycles between the writer any other transactions which conflicted with the reader (due to the reader's updates). This procedure repeats until the writer has been compared with all other transactions in the chain of dependencies. In this way, restrictions are propagated backward from the writer to transactions which are still in their read phase. Detection of a cycle causes the writer to abort. The reason for propagating restrictions is that cycles of higher degree could occur, involving the reader, writer and intervening conflicting transactions. Thus, the entire dependency graph must be checked. We present an example of a 3-way cycle in the section on correctness.

The case of an incoming write request conflicting with an already pending write request is handled similarly. The cc node first checks that both write requests are serializable. Then it checks for cycles between the incoming writer and other transactions which conflict with the pending writer, propagating restrictions where necessary. As with the locking algorithm, the updates of both writers must appear in the same order at all data servers where they conflict. Thus, the cc node may have to delay the second writer until the first writer sends a commit message.

A subtle point is that whenever a reader is restricted by a writer, the restrictions include not only the read and write sets of the writer but also the restrictions imposed on the writer, as well. Thus, restrictions are inherited as well as propagated. Again, the reason is that cycles among 3 or more transactions would otherwise go undetected.

Processing a termination message at the cc node (Figure 17) is fairly straightforward with one exception.

```
process commit or abort message
    for each rec in tran.readset
        remove tran from rec.readers
    for each rec in tran.writeset
        remove tran from rec.writers
    if abort message
        remove tran from sergraph
    else
        (* tran committed *)
        (* cannot remove tran from sergraph until
        ** all edges to and from tran are removed *)
        for each edge (tran,x) in sergraph
            remove edge
            if x has no edges
                remove x from sergraph
        for each edge (x,tran) in sergraph
            remove edge
            if x has no edges
                remove x from sergraph
        if tran has no edges
                remove tran from sergraph
```

Figure 17. Termination Message at CC Node: Non-Locking

Unlike the locking algorithm, we can not immediately remove the transaction from the serialization graph and forget its readset and writeset. The problem is that cyclic dependencies involving the transaction may occur after the transaction had terminated (for example, if an a read restriction it imposed on another is violated). If we removed the transaction from the serialization graph, the cycle would never be detected. Thus, we wait until a transaction has no connected edges before we remove it from the graph.

An interesting point is that it does not matter, from a consistency standpoint, if the cc node thinks that a request causes a conflict when it really does not. For example, suppose transaction T1 attempts to write a record but conflicts with transaction T2 which has read that record. Then T1 imposes restrictions on T2. If T1 subsequently aborts, the restrictions on T2 no longer apply. However, consistency is not violated if they are not removed. The worst that could happen is that T2 would abort unnecessarily. This implies that it is unnecessary to keep track of the "lineage" of restrictions which simplifies implementation of the cc node. This observation has been made in a different context in [Badal79a]. Note that this only applies to low conflict applications. Otherwise, the "phantom" conflicts could degrade performance by initiating unnecessary conflict analysis.

Correctness

In the locking algorithm, a transaction is not allowed to commit until the entire dependency graph is known. If a writer conflicts with a reader, the writer must wait until the reader declares its updates since the reader's dependencies are unknown until then. In the non-locking algorithm, we want to avoid waiting but we must still prevent cycles from developing. The restrictions, then, are constraints on nodes of the dependency graph which prohibit certain edges between transactions. If a writer conflicts with a transaction which is still in its read phase, we can allow the writer to immediately commit by restricting the reader from any action which would add a cyclic edge to the dependency graph. This suffices for cycles of length two. To prevent cycles of among more than two transactions, restrictions must be propagated and inherited. The following example illustrates a three-way cycle (see Figure 18).

This example requires some explanation. Transaction T1 reads record X and updates record Y, etc. Suppose that the requests are processed round-robin, i.e. T1 reads, then T2 and T3, then T1 writes, etc. Thus, the order of the requests is as shown for each record. At record X, for example, the T1 read preceded the T3 write. The order of the requests imposes constraints on the serialization order. Since transaction T1 tries to update record Y while

```
T1: read  X          T2: read  Y          T3: read  Z
    write Y              write Z              write X

Order of Requests

      X: T1,T3         Y: T2,T1          Z: T3,T2

Serialization Constraints

      T2->T1, T3->T2, T1->T3,    or equivalently

          T3   ->   T2   ->   T1   ->   T3


  Transaction:     T1             T2             T3
               ------------------------------------------
Restrictions: |
  no read     |    -              Y             Z {,Y}
  no write    |    -              X,Y           Y,Z {,X,Y}
```

Figure 18. 3-Way Deadlock and Inherited Restrictions

transaction T2 is a reader, T2 must precede T1 in any serialization. Similarly for the other transactions. Clearly, there is a cycle here. If restrictions were not inherited as described above, the cycle would not be detected. The list of restrictions is shown for each transaction. Note that T1 has no restrictions because it terminated first. T1 imposed read and write restrictions on T2. Specifically, T2 may not read record Y or update records X or Y. When T2 attempted a write, it conflicted with transaction

T3. Thus, it imposed restrictions from its read and write sets as well as its inherited restrictions (shown in braces for T3). Transaction T2 could then commit safely. When T3 attempted a write, it violated a write restriction so it was aborted. Had restrictions not been inherited, the cycle would not have been detected and T3 would have committed.

3.3. Robustness of Protocol

In the previous section, we assumed the existence of a perfect communications channel and that processors never crash. Here, we relax those assumptions and show that the passive scheme can tolerate such failures.

3.3.1. Lost Messages

Protocols for detecting and retransmitting lost messages are particularly important since the cc node must not lose conflict information. For example, if a read request is missed by the cc node but is processed by a data server, the cc node may not be aware of a conflict. Below, we consider how user nodes, data servers and the cc node can recover from lost messages. Note that in actual use, the Ethernet seems to provide a very reliable communications channel. A study of an existing 3 Megabit Ethernet found an error rate averaging one damaged packet per 6000 messages [Shoch79a]. Most errors were due to problems in

the receiver hardware, i.e., the packet was correctly formed on the cable. Further experimentation reduced this rate to one damaged packet per 2 million. In terms of 256 byte packets on a saturated Ethernet, this amounts to one error every 22 minutes. Errors would be even less frequent under lighter loads.

Messages Lost by User Nodes and Data Servers

When a user node is sent a data record in response to a read request, the reply serves as a built-in acknowledgment that the request has been heard and processed by the data server. However, a lost request can only be detected by lack of a response within a timeout period. Processing the read request involves an I/O operation at the server, so presumably, the timeout interval will be quite long. Thus, transmission errors will take a relatively long time to detect. Given that lost messages are infrequent, a long timeout and retransmission interval will not have much impact on overall performance. Another factor to consider is the burden that a long timeout interval places on the message subsystem. In order to retransmit a lost message, the sender must save a copy of the message and also record some state information. Because the timeout interval is long, this information accumulates and may cause performance problems if there is much message activity or if primary memory is scarce. For this reason, we assume that messages are explicitly acknowledged (as illustrated in Figure 19).

The message recipient sends an immediate acknowledgment before acting on the message. If the sender does not receive an acknowledgment within a short period after transmission, the message is resent. Version numbers can be used to detect and discard duplicate messages (e.g. if only the acknowledgment is lost). Note that sophisticated protocols to ensure that messages are received in the order sent are unnecessary here. Such problems arise in store-and-forward networks because of delays at intermediate nodes. In the Ethernet, messages travel directly from the sender to the recipient. As mentioned previously, since

```
user node                  data server

read req  -->
                           -->  get req
                           <--  send ack
get ack   <--

                           process read

                           <--  send data

get data  <--
send ack  -->
                           -->  get ack
```

Figure 19. Explicit Acknowledgment Protocol

most garbled packets are due to faulty receiver hardware, it makes sense to send explicit acknowledgments. The error will be detected immediately, not after a long timeout delay. This reduces the amount of state information that must be maintained for long periods at the sender.

Thus, messages lost by user nodes or data servers will be detected after a short timeout interval and the message will be resent. Note that explicit acknowledgments do not obviate the need for timeouts. The acknowledgment only tells the sender that the message has been heard and placed on an input queue. For example, a data server may crash after acknowledging a read request and this must still be detected by a timeout at the user node.

<u>Messages Lost by CC Node</u>

The cc node is concerned only with three types of messages: read requests, write requests and termination messages (commit or abort). Further, we assume that write and termination messages are broadcast (strictly speaking, these messages are multicast since they are sent only to the data servers which process requests for a transaction). The problem is that the cc node may miss a read or write request and thus, lose conflict information. One solution is to have the cc node also send acknowledgment messages when it overhears a request. However, this adds an overhead of one additional message per request and slows down the cc node and the data servers. Our solution is to have the cc node eavesdrop on acknowledgment messages as well as request messages. In this way, the cc node can tell if a data server has actually received a request and intends to process it. But, we have a slight problem if the cc node hears an acknowledgment which is not heard by the user node. In this case, the user node will resend the request and the data server will resend an acknowledgment. We assume that the data server and the cc node can detect such duplicate request messages.

This solves the problem of the cc node hearing a request that is not heard by a data server or a user node. But we still have a problem if the cc node misses an acknowledgment or a request and an acknowledgment. This is essentially an unrecoverable error. If a data server begins processing a request which the cc node has not heard, then conflict information has been lost. Our solution is to detect lost request messages and abort the transaction (a bit harsh, but, again, errors are infrequent). Lost data request messages can be detected by including a count in each data request which gives the cumulative number of read and write requests made by the transaction. The cc node maintains a separate count for each transaction and compares its count with the count in each read and

write request. If the counts do not agree, the cc node must have missed a request and the transaction is aborted.

Note that it is not possible to recover from the error by having the cc node ask the data server for conflict information. The data server could provide local conflict information about the request but what is really important is the global sequence in which requests occurred. The cc node must know not only what conflicts result from an access, but in what order requests occur in order to serialize the transactions.

We now consider the effect of the cc node missing a termination message from the user node. This problem can be attacked in two ways. First, we could require that the user node receive an explicit acknowledgment from the cc node for termination messages. Using the Acknowledging Ethernet (described below), this scheme imposes very little overhead. Alternatively, we could detect a missed termination message with a timeout. The difficulty here is that the the cc node may assume conflicts are caused by the "unterminated" transaction during the timeout period. However, as mentioned above, phantom conflicts never violate consistency so the delay (before detection) is not a real problem.

Acknowledging Ethernet

In order for the cc node to keep track of acknowledgment messages, it must maintain state information for each message sent. In a heavily loaded system, this state information can become a burden. An Ethernet acknowledgment protocol has been devised which can eliminate the need for much of this information [Tokoru77a]. In this scheme, an intelligent Ethernet interface is used to send an automatic acknowledgment as soon as a message is correctly received. Thus, the sender receives an immediate acknowledgment that the message has arrived. If no acknowledgment is heard, the message is immediately resent. The protocol requires that when a message is ready to be transmitted, the sender waits until the transmission medium is free, and then waits another "slottime" (the end-to-end signal propagation time). If the channel is still free, the message may be sent. The purpose of waiting an extra slottime is to yield to any acknowledgment packets from the previous message. Because an acknowledgment packet is short (16 bytes), the protocol does not degrade performance. In fact, an analysis in [Tokoru77a] showed that because errors are detected immediately, the scheme can provide better response time and higher channel utilization than the original Ethernet.

The cc node may take advantage of this protocol because it eliminates the need to maintain state information for long periods. The cc node only has to buffer one message at a time. If an acknowledgment is not heard immediately, the message is ignored under the assumption that it will be resent. However, the protocol, as presented in [Tokoru77a], does not consider broadcast messages. This is unfortunate since we assume that write and termination messages are broadcast. But, a simple extension may be made to incorporate a limited form of broadcasting. Suppose a message is to be sent to a subset of nodes. This has been referred to elsewhere as multicasting and has been supported in previous distributed systems (see Farber73a).

We suppose destination addressing may be done with a bit map in the message header or simply a list of site identifiers. The destination addresses can be used to impose a linear ordering on the recipients (e.g. ordinal position in the list of recipients). This ordering can then be used to synchronize transmission of acknowledgment packets. For example, the first recipient sends an acknowledgment during the first slottime. The second recipient sends an acknowledgment during the second slottime, etc. By chaining acknowledgments in this way, all recipients of a multicast message can send an immediate acknowledgment to the sender. If recipient 'i' misses the message, it will not respond and the sender only gets i-1 consecutive acknowledgment messages. Since the chain is broken, later recipients (numbered higher than i) may be unable to send acknowledgments. This is because another site may have a message to send and will begin transmitting when the Ether is free for a slottime (due to the missing acknowledgment). If this happens, later recipients (i.e. after site i) can detect it and will simply wait for the sender to retransmit the message. Note that the retransmission is only addressed to those sites which did not respond, not all sites. Thus, if a message is correctly received with positive probability, the protocol eventually terminates.

This extension to the acknowledging Ethernet protocol allows the cc node to detect lost write and termination messages as easily as read requests. When a multicast message is heard, it simply monitors the acknowledgments and notes which sites respond. No long term state information need be kept. This would probably have to done in hardware (as part of the cc node Ethernet interface board) however, because of the real time constraints.

3.3.2. CC Node Crash

The cc node is the repository of all conflict information in the system. Consequently, if it crashes, the entire system is disabled. A failure of the cc node may be detected by a timeout at a user node. If the cc node fails to grant a transaction permission to terminate the user node will attempt to contact the cc node. The cc node may have simply missed the write request. If the cc node has actually failed and no contact can be made, recovery measures are invoked as outlined below.

The simplest technique is to just run several cc nodes in parallel. One cc node would designated as the primary cc node and the others would be secondary cc nodes, running in the background. The reason for the primary node is that multiple cc nodes may reach different conclusions about a transaction. For example, one cc node may decide to abort a transaction because it missed a request, although all the other cc nodes may have heard the request and allowed the transaction to continue.

This implies that secondary cc nodes must monitor the messages of the primary cc node to determine if the primary site has aborted any transactions due to lost messages. Also, if a secondary cc node misses a request, rather than abort the transaction it must correct its conflict information by contacting the primary cc node. If the primary cc node fails, the secondary cc nodes nominate a new primary cc node and continue. When a failed cc node has recovered and is ready to restart conflict analysis, it must contact another cc node to obtain a copy of the conflict information.

A problem occurs if the primary cc node fails while resolving an inconsistency (due to a missed message) with a secondary cc node. In this case, the cc node may not know the true state of the transaction. But, recovery is simple since the secondary cc node can always contact the user node to discover if the transaction has committed or aborted. The user node will always know the state of the transaction. The source of these problems is that the multiple cc nodes have their own communications interface. Perhaps a better solution is to have all cc nodes share a single message interface. In that way, all cc nodes receive or miss a message together.

An alternative to redundant cc nodes is to have the user nodes elect a new cc node when the cc node fails. This requires that data server nodes maintain some conflict information, in particular, the local dependencies and the order of reads and writes on each data object. Using this information, a new cc node could reconstruct the global conflict information and processing could begin again.

This idea has been developed and made robust with respect to multiple site failures in [Menasc80a].

A third technique for continuing processing after a cc node failure is to switch to a distributed concurrency control algorithm. If we assume that data servers maintain access logs as described above, it would be easy to switch to distributed concurrency control once the centralized system failed. The system could run with the distributed algorithm, perhaps at reduced rates of throughput, until the cc node had recovered. A distributed concurrency control algorithm is described in detail in Section 4.1.1. Below, we outline the actions of the user node and cc node when converting from the passive scheme to a distributed algorithm and back.

We assume that data servers maintain access logs for all active transactions. This redundant conflict information is already required for the passive locking algorithm (to block conflicting read request). It is needed in order to transparently switch from a passive scheme to a distributed scheme. We assume that a user node detects the failure of the cc node when the cc node fails to grant permission to commit within a timeout period. Presumably, the user node will make several attempts to re-establish contact before assuming that the cc node has crashed. When all such attempts fail, the user node switches to a distributed algorithm as outlined in Figure 20 (we show only the locking algorithm). The user node obtains a copy of the access log for every record it has updated. Using this information, the user node can determine if the updates caused any conflicts and construct the serialization graph as in the passive locking algorithm. In order

```
when detect failure of cc node
    cctype := distributed
    initialize sergraph to null
    for each record in writeset
        get access log from data server
        for each conflicting reader in access log
            add edge (rdr,tran) to sergraph
        for each conflicting writer in access log
            add edge (wrtr,tran) to sergraph
   loop:for each edge (x,tran) in sergraph
            request sergraph of transaction x
               from user node for transaction x
            merge x.sergraph with tran.sergraph
        if cycle in sergraph
            abort tran
        if new nodes added to sergraph
            go to loop

when receive request for serialization graph
    if cctype = passive then
        attempt contact with cc node
        if no contact
            cctype := distributed
            determine sergraph as above
        else
            wait for permission to commit from cc node
            reply with null sergraph
    else
        reply with sergraph
```

Figure 20. Passive to Distributed Switch at User Node

to detect global deadlock, however, the user node must construct the global serialization graph. Thus, it must contact the user nodes of conflicting transactions and requests their parts of the serialization graph. Newly initiated transactions at the user node use distributed concurrency control until the cc node recovers (as described below).

When a user node receives a request for its serialization graph, it may be unaware of a cc node failure and still be executing with passive concurrency control. Alternatively, the cc node never failed or failed and recovered unbeknown to the requestor. Thus, a request for a sergraph is followed by a check to determine if the cc node has actually failed. If cc node is working, the user node waits for permission to commit and then responds with a null (i.e. no more conflicts) sergraph.

We now describe how a cc node re-enters the network and restarts conflict analysis (see Figure 21). The cc node begins by eavesdropping as before and informing the user nodes that is has recovered. But it cannot begin immediate conflict analysis because it does not know the conflict history of the currently active transactions. In addition, the active transactions may be operating under a distributed algorithm. The protocol we adopt is that the user nodes respond with the serialization graph for the

```
cc node recovery
    clear all data structures
    begin to monitor data requests
    broadcast "live cc node" to user nodes
        and obtain tran.id and conflict history
        of currently active transactions
    newly initiated transactions use passive cc node
```

Figure 21. Distributed to Passive Switch at CC Node

active transactions and the user nodes return to passive concurrency control.

Note that a difficulty occurs if the cc node crashes and recovers before a user node detects the failure. In this case, the user node may detect the crash when the cc node broadcasts the "live cc node" message. If it misses the broadcast message, however, the error can still be detected. Recall that request messages include access counts for each transaction. The cc node will detect the inconsistency between its access count for the transaction and the access count at the user node. The transaction may then be aborted, or the cc node may elect to derive conflict information from the data servers itself.

3.3.3. User Node or Data Server Crash

A user node crash would be detected if the cc node did not hear a request or termination message for the

transaction after a timeout period. In the simplest case, a transaction is still in its read phase. In this case, the cc node may simply broadcast an abort message on behalf of the transaction. Similarly, if the cc node has decided to abort but has yet to hear a termination message, the cc node may also send an abort for the transaction. The worst case occurs when the cc node has already given the transaction permission to commit but has not yet heard its termination message. It is difficult to determine if the transaction has actually decided to commit or abort. For example, a failed data server or human intervention could also cause the transaction to abort. In addition, the abort (or commit) message may already have been sent by the user node but the cc node managed to miss both the termination message and all acknowledgments (assuming the cc node does not explicitly acknowledge termination messages). One option is to poll all data servers to determine if any decision had been reached. In the absence of any information, the only choice is to wait for the user node to recover. Presumably, part of the recovery procedure at the user node will be terminating the transaction.

A failed data server causes the cc node to abort all uncommitted transactions active at the server.

3.4. Performance Considerations

Previous studies of centralized concurrency control algorithms have shown that under certain conditions, the central node can become a bottleneck and slow throughput [Garcia79a, Ries79a]. Garcia-Molina found that when the amount of state information required to maintain conflict information is too large for primary memory, then paging occurs which increases response time. In his particular case, data records were timestamped. Checking the serializability of an access required a comparison of the transaction timestamp and the data record timestamp. It was assumed that primary memory could not hold the timestamps of all data records. Ries found that under heavy loads, processor contention at the central site caused performance problems. In his model, however, the central site managed a portion of the database as well as performing conflict analysis. This, undoubtedly, contributed to the processor load and, in fact, he found that when the database functions were removed, performance improved considerably. Note that both models assumed that transactions preclaimed their resources which we do not. Thus, we would expect that runtime claiming of records might cause even more performance problems.

We claim that for the range of loads we are interested in, our passive scheme will not become a bottleneck. For

example, assume the cc node is a 2 MIP processor and the system is designed to process 100 transactions per second. Note that this is a rate which has never been achieved by a commercial system. A system offered by Tandem Computer Corporation does well to execute 10 transactions per second [DeWitt81a]. We further assume that each transaction reads 3 records and updates 1 record. This amounts to 5 messages per transaction (3 reads, 1 write, 1 termination message) which must be processed by the cc node or 500 messages per second. Note that the Ether acknowledgments are not counted since they are done in hardware and are essentially transparent to the transaction. This allows 2 milliseconds (or 4000 instructions) to process a message without queueing delays. In this time, the cc node must retrieve the message and update data structures for the transaction, the record and the global conflict graph. Based on measures of existing systems, a commonly accepted "cost" to send a message is 5,000 instructions. Thus, it might seem as though the cc node is too slow.

Much of the cost in sending a message is due to operating system overhead. In some schemes, a message must percolate down through seven layers of an operating system before it is ready to be transmitted [Tanenb81a]. However, some recent work has been done in directly microcoding operating system functions. In particular, encoding message primitives can reduce the cost of sending a message to approximately 50 microseconds for certain classes of messages [Specto80a]. Considering that the cc node is a dedicated processor, it seems reasonable that it would have a specially tailored operating system. Using such features as microcoded message routines, we feel that a system can be easily devised to process messages well within our time constraints.

Message processing is not the only potential problem. We must also eliminate or severly limit the amount of I/O done by the cc node. There are two potential sources of I/O at the cc node: recovery protocols (i.e. writing log records to stable storage) and paging I/O due to insufficient primary memory. It is not even feasible to perform one disk write per transaction. For example, assuming 100 transactions per second, the cc node would have to do the write in 10 milliseconds. This is well under the average 30 millisecond access time typical of large capacity disks [IBM80a]. Thus, it is clear that the cc node cannot do any recovery since writing a log record would consume too much time.

We must also guarantee that the amount of state information maintained for all active transactions and data records does not exceed the size of primary memory. This would cause paging I/O which would also severely degrade

performance. However, not much information needs to be maintained for each active transaction and each active record. For each record, only the active readers, writer and the wait queue are needed. For each transaction, only the read set, write set and conflicting transactions need to be kept. The data structures all contain either transaction identifiers or record identifiers which presumably are very short. Assume that it takes 500 bytes to maintain information about an active data record. And assume that an active transaction requires a 2500 byte data structure (a high estimate, especially for small transactions). If we have one megabyte of primary memory to divide equally among each type of data structure, we can support 200 active transactions and 1000 active records. This is well within our requirement of 100 transactions per second. And, even doubling the amount of primary memory would still be within the capacity of existing mid-size computers (such as a Vax). Thus, the cc node would have have very low requirements for I/O.

3.5. General Applicability of Eavesdropping

As presented above, the passive concurrency control scheme has limited applicability. It is constrained to run in an Ethernet-like environment. In this section, we generalize the passive idea and apply it to other architectures and other environments.

Non-Ethernet Architectures

The only real constraint our scheme places on an architecture is that it have a broadcast capability. Thus, our scheme would work well in token ring networks (such as DCS, Farber73a) with the following provision. A common acknowledgment protocol in token rings is to have the message recipient set a bit as the end of the message passes by. However, unless the cc node lies between the sender and recipient, it will not overhear the acknowledgment. Thus, an explicit acknowledgment protocol is needed to guarantee that the cc node overhears the message as well as the acknowledgment. One possibility is to give the cc node control of the token after a message has been transmitted. The sender will hear the acknowledgment but will pass it on. When the cc node hears the acknowledgment, it removes it from the ring and generates a new token.

Another processor interconnection scheme that is becoming popular is a broadband bus. Conceptually, this is similar to an Ethernet, except that rather than transmitting messages on a single frequency, the bus can carry messages on multiple channels simultaneously. Thus, bus interfaces are really radio transmitters and receivers. The problem here is that the cc node can only monitor one channel at a time, while requests may be sent on various channels. The obvious solution, then, is to dedicate one

channel to the control messages of interest to the cc node, i.e. read, write and termination messages. However, write messages can get rather large since they contain data records (read requests and termination messages contain no data). Another possibility, then, is to equip the cc node with multiple receivers and dedicate a separate channel to write messages. A real problem with the passive scheme here is not eavesdropping on the network. It is the large capacity of the communications bus itself. Such a bus could support many more processors than an Ethernet. It is not clear that a single cc node could keep up with the traffic. Similar problems would occur with a light-pipe communications bus.

A satellite broadcast network is a natural fit for the passive concurrency control scheme. A ground station would be capable of monitoring all messages relayed by the satellite. However, a general loosely-coupled network, such as ARPANET, would present problems due to the store-and-forward nature of message transmission. Some efficient broadcast schemes have been proposed for such networks (for example, Dalal78a). However, it seems unlikely that a business application could ever perform well in such an environment.

Inter-Gateway Transactions

Due to bandwidth limitations and propagation delays, an Ethernet is limited both in the number of nodes it can support and in the maximum length of the cable. For large applications (e.g. connecting several buildings), it may be necessary to have separate Ethernets connected via gateways [Metcal76a]. Such a scheme presents problems for our passive algorithm because a single cc node can only monitor traffic on one Ethernet. This makes conflict analysis difficult for transactions which access data across networks (i.e. inter-gateway transactions). In this case, the conflict information is distributed across several cc nodes instead of several data servers. There are several possible solutions to this problem. If there are just two networks connected by a gateway, the cc node could be attached to the gateway and thus be able to monitor both networks. If there are several networks, we have the classical distributed concurrency control problem, except the conflict information is distributed at cc nodes, instead of processing sites. Thus, one of the distributed concurrency control schemes discussed previously would have to be used. Note, that the passive scheme could be used at a higher level. For example, the cc nodes could be connected with their own distinct Ethernet in order to resolve conflicts among inter-gateway transactions. This network

could have its own cc node which has the sole purpose of resolving global conflicts. Presumably, inter-gateway transactions will not be very frequent.

Replicated Data

The discussion on passive concurrency control assumed that data was not replicated. Replicating data presents no difficulties for our algorithm, however. The basic question is which data server will respond to an access request for a replicated record. It does not matter how this decision is made so long as the cc node and the data servers use the same rule.

Passive Recovery

An extension of the passive concurrency control idea is to have a passive recovery node as well. Rather than giving the data servers responsibility for database recovery, a dedicated node could perform this function. This reduces the overhead at all data servers since they no longer have to perform log writes. Recall that the two-phase commit protocol requires at least two log writes per transaction (one "ready" log record, one termination log record). A central recovery node would eliminate these log writes at all data servers and potentially result in higher throughput. In this scheme, the data servers are nothing more than intelligent disk controllers, processing read and write requests as they arrive.

The difficulty with passive recovery, of course, is the inability of the recovery node to keep pace with requests. A load of 100 transactions per second requires at least 200 log writes per second or an access time of 5 milliseconds. As mentioned earlier, a typical disk speed is 30 milliseconds per access, so a centralized recovery node would not be able to keep up. One possibility is to have multiple disks per recovery node. Another possibility is to use a different type of nonvolatile storage media to maintain the log file. Magnetic cores memories have microsecond access times. Bubble memories have access times on the order of milliseconds. Both are recoverable after a processor failure, but are expensive. It may be possible to cache log writes in small scale bubble or core memories until they may be migrated to disk. A third alternative is to "batch" together log operations for separate transactions. In this way, the cost of the log operation is amortized over several transactions. The trade-off is that some transactions are delayed while the recovery node waits for enough transactions to request commit. However, in a heavily loaded system, this delay would not be long because so many transactions are active. And in a lightly loaded system, there is no need to batch log operations in the first place. This solution warrants more

investigation. To the best of our knowledge, this is the first time it has been proposed.

CHAPTER 4

PERFORMANCE ANALYSIS

In this chapter, we describe the simulation model and some experiments used to evaluate the performance of the passive concurrency control algorithm. The purpose of the experiments was twofold: to see if the passive scheme performed well under a variety of conditions; and to see if the passive scheme performed better than corresponding distributed algorithms. Four simulations were written: a distributed locking algorithm, a distributed non-locking algorithm and passive locking and non-locking algorithms. In this section, we describe the simulation model in detail, including the distributed concurrency control algorithms used for comparison. Then we discuss a set of concurrency control experiments which compared our centralized passive scheme with the distributed algorithms. The final section is a critical examination of the simulation and its results and points out problems and areas for further research.

4.1. Simulation Model

There are three traditional approaches to performance analysis: analytic, simulation, or benchmark. Analytic evaluations are done at a rather gross level of detail because of simplifications required to keep the formulae

tractable. System benchmarks are the most accurate and the most detailed. However, they require an existing system. The accuracy of a simulation varies depending on the level of detail simulated. Based on previous analysesRies79a, Garcia79a, we do not expect differences between concurrency control algorithms to result in "order of magnitude" differences in throughput. Since we expect rather fine variations, we chose to construct a fairly detailed simulation model for performance analysis.

4.1.1. Structure of Model

An issue in our study was precisely which components of a distributed database management system should be modeled. We decided to ignore any aspects that were not directly related to transaction management. Therefore, each simulation modeled a concurrency control algorithm and a recovery protocol only (the recovery protocol was always distributed two-phase commit). We did not consider the cost of schema access or index operations. Also, we did not concern ourselves with data distribution issues. By doing this, we felt that any differences observed in the results could be directly attributed to differences in the concurrency control algorithms. Thus, any numbers presented for transaction response time or system throughput should be interpreted in a relative terms, not in an absolute sense. The next few paragraphs describe the structure of the simulation model, including the resources modeled, the input variables and response measures. For clarity, at the end of this section we include an example which illustrates how transactions are executed in both the distributed and passive approaches.

<u>Processors</u>

All processors were identically configured in the simulation (i.e. one disk, one communications interface and a central processor). However, the central processor speed and message processing overhead were not the same for each processor type. Since the cc node and the data servers perform a dedicated function, we assumed they were provided with fast hardware and a specially tailored operating system for better performance. Thus, in the experiments, the data servers and the cc node were usually given faster processors than the user nodes. Also, the operating system overhead required to send or receive a message (formatting, packet assembly/disassembly, etc.) was usually higher for the user nodes than for the cc node and data servers. The simulation model included queues at the central processor, the disk, and the communications interface and service was provided on a first-come, first-served basis. The time to access the disk was fixed at 30 milliseconds and I/O was done asynchronously. Finally, for the sake of simplification, we assumed that processors never fail. We feel that

in a real system, processor failures will not be so frequent as to impact performance. Thus, nothing would be gained by simulating processor crashes and associated recovery algorithms. However, the cost of protocols which make recovery possible can affect performance. Thus, the two-phase commit protocol was modeled in the simulation.

A <u>user node</u> supervises execution of one transaction at a time. When a transaction terminates, another one immediately begins at the user node. Thus, transactions are constantly queued at user nodes. The number of active transactions (and thus the number of user nodes) is a simulation parameter but is constant for the duration of a run. A user node maintains a workspace for its transaction which contains all records read and written. We assume that a transaction's updates are sent to the data servers with a single broadcast message. The write message also initiates conflict analysis and the distributed two-phase commit protocol. When both conflict analysis and the distributed two-phase commit protocol have been completed, the user node broadcasts a commit or abort message, the transaction terminates, and a new transaction begins.

<u>Data servers</u> process read and write requests sent by user nodes. Note that data servers observe the two-phase commit protocol of writing two log records (a "ready" and "commit" record) before updating the database for a

transaction. We assume that data servers also maintain a workspace for any transaction which has sent it a write request. This is a small, short-term workspace in which updates are temporarily held until a termination message is received. Thus, the workspace exists only for the duration of the two-phase commit protocol (i.e. between reception of the write request and the commit or abort message). Also, we assume that data servers have a large number of data buffers in which "active" records may be cached. A record remains active as long as it is in use by a transaction. A read request for an active record is served from the buffer and this reduces I/O. Similar caching strategies are used in most operating systems. As described below, in the distributed concurrency control algorithms both the data servers and user nodes share the task of conflict analysis.

The <u>cc node</u> is responsible for passive conflict analysis. To do this, it eavesdrops for data requests on the Ether. An important point is that a write request from a transaction also serves as a request to commit. When the cc node determines if the updates are serializable, it sends the transaction a message telling it to commit or abort (although the actual termination message is sent by the user node). These are the only messages sent by the cc node. We assume that all conflict information needed by the cc node can be maintained in primary memory. Thus, the

cc node does no I/O. This implies that only central processor contention could cause the cc node to become a bottleneck.

<u>Messages</u>

Modeling of the communications medium was fairly detailed. It included collision detection and use of the binary exponential backoff algorithm [Metcal76a] for selecting a retransmission interval. Also, the Acknowledging Ethernet (Tokoru77a) was simulated rather than standard Ethernet. This protocol makes it easy for the cc node to keep track of acknowledgments. Thus, it was not necessary to model message state information in the cc node. Finally, the Ethernet was modeled as a perfect channel, i.e. messages were never lost. As with processor failures, it was felt that transmission errors would be so infrequent as to not impact performance. Measurements of an actual Ethernet [Shoch79a] have shown that, indeed, errors are extremely rare (1 error per 2 million packets). Note that although we did not model actual failures, the simulation did include the overhead cost of dealing with such errors (e,g. the Acknowledging Ethernet protocol).

We assumed that messages of the same type were the same size (e.g., all read requests were the same length). The size of messages ranged from 100 bytes for a simple control message (e.g. commit transaction) to 500 bytes for a conflict analysis message (e.g. dependency graph). The size of messages containing data records (read replies and write requests) were larger by the length of data records (1000 bytes). Acknowledgment packets were 16 bytes long and were sent immediately after a message was received as per the acknowledging Ethernet protocol.

Database

The database was evenly partitioned among the data servers and there was no replication. The record length was fixed at 1000 bytes. Also, the access pattern was random (all records were equally likely to be accessed). Note that we did not vary the size of the database as has been done in other simulations [Ries78a, Garcia79a]. The primary result of varying database size is to increase or decrease the number of transactions which conflict (since database size affects locking granularity). But, we elected to vary the amount of contention directly by fixing the probability that a request will conflict (see Section 4.1.2). If a request should conflict (decided by a Bernoulli trial), an already active record was randomly selected. Otherwise, an inactive record was chosen and made active. Thus, the choice of a database size was immaterial, so long as it was large enough to provide concurrently active transactions with as many non-intersecting

read and write requests as was necessary.

Transactions

Transactions were homogeneous in that they all read and updated the same number of records. The transaction size was a simulation parameter. Also, we assumed that the writeset was a random subset of the readset, i.e. a transaction only updated records it had read. Given the underlying application (a business database) we felt this was a much more reasonable assumption than the other extreme, i.e. records to be written are chosen randomly from the entire database. This assumption was only made in order to make the simulation more realistic. It does not constrain the concurrency control control mechanism, unlike other schemes which also make this assumption (Bernst78a, Thomas79a).

Transaction execution consists of two phases: a read phase and a commit phase (see Figure 22). The read phase loop is executed once for each record in the readset. The "thinktime" delay simulates the processing time needed to process the previous record and to formulate a new request. This delay is fixed at 500 instructions for all simulations. This number is rather arbitrary. Increasing it simply decreases the number of transactions simultaneously active at the data servers and increases transaction execu-

```
Read Phase                      Commit Phase

loop                            choose writeset from readset
   delay for "thinktime"        send updates
   pick record randomly         await log acknowledgments and
   send request                    do/await conflict analysis
   await reply                  send commit or abort message
end
```

Figure 22. Phases of Transaction Execution

tion time. We feel that 500 instructions represents a reasonable delay for inter-request processing for an optimized batch transaction. After the delay, a record is randomly selected in accordance with the conflict rate. The read request is then sent and the transaction waits for the data record to be returned.

The commit phase begins with the selection of the writeset and sending the updates. During this phase, two separate sub-tasks must be completed: conflict analysis and the distributed two-phase commit protocol. Thus, the term "commit phase" may be misleading since the phase includes not only two-phase commit, but also conflict analysis and the possibility that the transaction may abort. In the distributed concurrency control schemes, the user nodes actively participate in conflict analysis by exchanging messages with conflicting transactions at other

user nodes. In the passive schemes, however, the user node simply has to wait for a commit or abort decision from the cc node. When a decision has been made and the data servers have completed the first phase of the two-phase commit (and responded "ready"), the termination message is broadcast by the transaction node to the data server nodes.

Distributed Conflict Analysis

In the distributed concurrency control algorithms, the user nodes and data servers share responsibility for conflict analysis. In the locking scheme, data servers maintain read and write locks on data records. When a read or write request is blocked, the data server sends a conflict message to the user node which issued the request. The purpose of a conflict message is to inform the requesting transaction that it is blocked. The blocked transaction must then contact the blocking transactions in order to check for deadlock and guarantee serializability (i.e. construct the global dependency graph).

In the distributed non-locking algorithm, the roles of the data server and user node are similar. Conflict messages are sent by a data server whenever a read request overlaps with an active writer or a write request overlaps an active reader. However, upon receiving a conflict message, user nodes exchange restriction lists as well as portions of the dependency graph.

In both the locking and non-locking distributed algorithms, the distributed two-phase commit protocol is simulated for recovery. Note that it would have been possible to give the data servers complete control over conflict analysis. From a correctness standpoint, it does not really matter which processors maintain and exchange the dependency graphs and restrictions lists. However, the data servers must continually process requests while the user nodes are essentially idle between requests. Thus, it seemed a good idea to off-load the concurrency control function to the user nodes to reduce the load on the data servers.

Example Execution

We illustrate below a sample execution of both the distributed and passive locking algorithms. In the examples, when transaction T1 attempts to update record X, it conflicts with transaction T2 which has read X. It may be assumed that T1 and T2 read and update other data records without conflict. In both figures, processors are enclosed in boxes and messages sent and received are shown in time order, top to bottom. We show only the messages pertinent to the conflict. Also, messages required for conflict analysis are underlined. The distributed locking algorithm

is shown in Figure 23. When T1 sends its update request for X, the data server responds immediately with a conflict message informing T1 that its write is blocked by the read lock of T2 ("cnfl msg"). To check for deadlock, T1 must then ask T2 for its portion of the dependency graph ("dep req"). T2 responds when it has computed its part of the graph ("dep graph"). Note that because this is a locking algorithm, T2 will not respond until it has sent its updates and also received conflict messages from the data servers. In the meantime, the data server for X has written the log record for T1 and responded. When T1 receives the dependency graph from T2 and determines that there are no more conflicts, it may commit. Note that resolution of the conflict required 3 messages.

The passive algorithm is illustrated in Figure 24. The cc node is immediately aware of conflicts because it eavesdrops for request messages on the Ether. Recall that a write request is also a request to commit. When the cc node determines that there are no cycles in the dependency graph, it grants the transaction permission to commit ("com ok"). The transaction must wait for the cc node to respond and for the data servers to write their log records. Then it sends a commit or abort message, as dictated by the cc node.

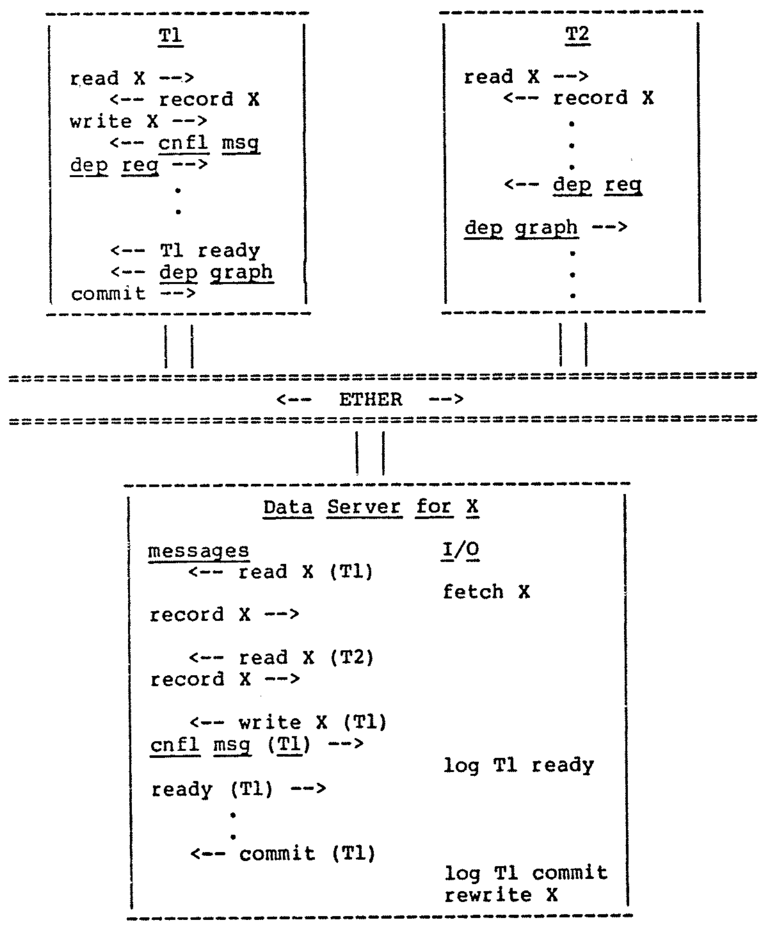

Figure 23. Distributed Locking Algorithm

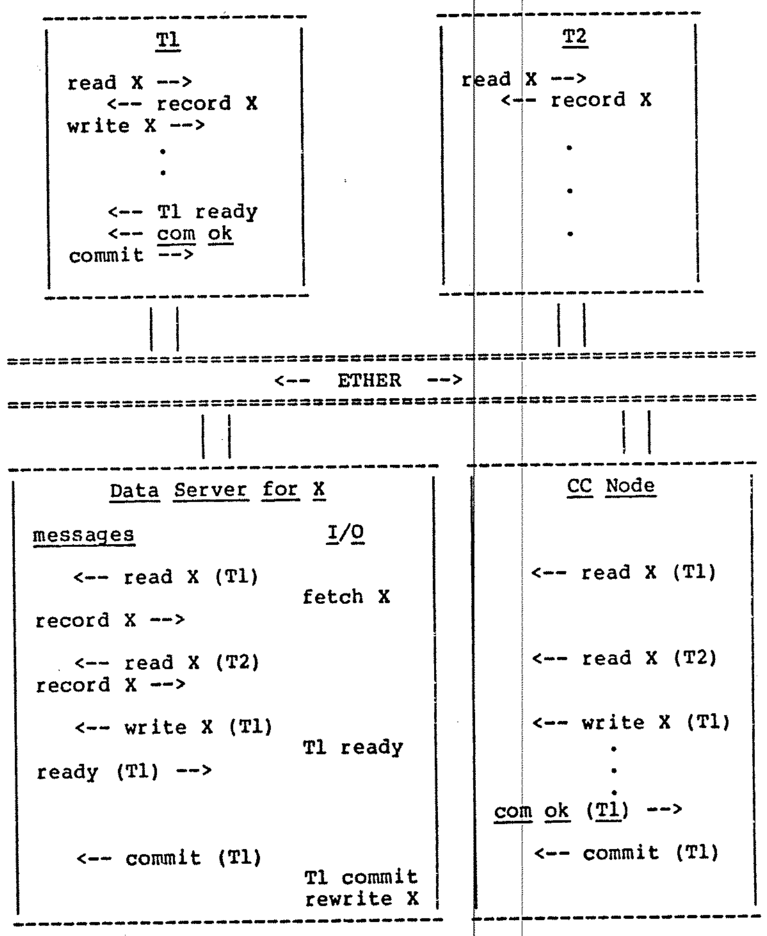

Figure 24. Passive Locking Algorithm

Execution of the non-locking algorithms is similar in structure to the locking algorithms. In the distributed non-locking scheme, instead of requesting a dependency graph, T1 sends T2 access restrictions. Thus, T1 need not wait for T2 to determine its dependencies. In the passive non-locking scheme, the cc node can immediately decide to commit or abort T1 by checking the global dependency graph and imposing restrictions on T2. Again, the wait time is reduced.

4.1.2. Independent Variables

Four variables were considered of primary importance in the simulations. They are shown in Figure 25. The message overhead variable specifies the number of instructions required by the operating system of a user node to process a message. The message overhead for the cc node and data servers was fixed at 1500 instructions for most experi-

Variable	Range	
message overhead	500, 2500, 5000, 7500	(instructions)
conflict rate	0.0, 0.1, 0.2, 0.3	(prob. cnfl.)
transaction load	4, 25	(user nodes)
bubble memory log	no, yes	

Figure 25. Primary Independent Variables

ments. Although 1500 instructions may seem low, recall that these processors do not require general purpose operating systems. This fact permits certain optimizations which can reduce overhead. Recent work in this area is encouraging; the overhead for sending datagrams has been reduced to approximately 50 microseconds [Specto80a].

The conflict rate is the probability that a read request will access an already active record (i.e. it is the probability that a read conflicts with another read or write request). For example, a conflict rate of 1.0 would cause all transactions to read the same data record. This may seem a rather artificial way of modeling contention. Perhaps a better way would have been to alter the access pattern to a non-uniform distribution. However, we wanted to observe how the number of conflicts affected the performance of each algorithm. To do this, it makes sense to vary the conflict rate directly. Note that there is no corresponding probability variable for write requests because the writeset is a subset of the readset. The number of concurrently active transactions is specified with the transaction load parameter. Since a user node runs only one transaction at a time, this also corresponds to the number of user nodes on the network.

Early in our experiments we noticed that the performance of all algorithms was affected by the distributed two-phase commit protocol. One reason was that the log file was maintained on the same device as the data. The bubble memory log variable specified that the log file was to be maintained on a bubble memory device. This device was not only separate from the data disk but was faster as well. The bubble log access time was set at 3 milliseconds compared with 30 milliseconds for the disk. This is consistent with speeds quoted for such devices in the literature [Teraji80a]. Note that magnetic tape could also be used to maintain the log file (although at a slower access time than bubble memories). However, if a transaction aborts, it is more difficult to undo a transaction from a tape log than a disk log. This is because the log file must be reread to undo a transaction and tapes are serial devices. This implies that the log tape must be held exclusively during this period so it can be rewound, searched and repositioned [Gray78a]. Thus, database activity is effectively halted until the transaction is undone. And there are other situations where the log file must be reread (for example, recovery or a request for the status of a transaction). The lack of concurrent access to tapes suggests that random access memories are a better choice for log files than serial memories.

In addition to the primary variables, there were secondary variables which had indirect and minimal effects

on performance. These are summarized in Figure 26. The most common settings of the variables are specified first. Experiments were also conducted with the variables set to the second values. The bandwidth of the Ethernet was specified with the Ether capacity variable, in megabits-per-second. The number of data records read and written by a transaction was set with the transaction size variable. The cpu speed variables specified the relative speeds of the user node, data server, and cc node processors in units of millions-of-instructions-per-second (MIP).

4.1.3. Dependent Variables

The dependent variables of interest are summarized in Figure 27. Commit time refers to the amount of time spent in the commit phase (between sending updates and sending a

Variable	Range
Ether capacity	3 megabits per second 10 megabits per second
transaction size	3 reads, 1 write 6 reads, 2 writes
cpu speed	0.5 MIP user nodes, others 2 MIP all 1 MIP

Figure 26. Secondary Independent Variables

primary
- commit time
- throughput

secondary
- Ether load
- server queue length
- number of messages sent

Figure 27. Primary and Secondary Dependent Variables

termination message) by transactions with conflicts. The justification of this measure is that all algorithms are essentially identical during the read phase. However, at the start of the commit phase (i.e. when the updates are sent) conflict analysis begins and the algorithms differ. By considering only transactions with conflicts we get, in effect, a measure of the cost of conflict resolution. The throughput measure reflects the impact the concurrency control algorithm has on transaction throughput. For this value, the response times of all transactions are observed (regardless of whether or not they have conflicts). Response time is defined as the average time between initiating a transaction and sending a termination message. Throughput is defined as the inverse of the average response time multiplied by the number of user nodes.

The secondary measures reflect the accuracy of the simulation (e.g. indicate if the system is balanced) and provide additional information which helps provide insight into the results. The Ether load is the utilization of the bus capacity. If the load is too high, the simulation becomes a model of Ethernet contention, rather than measuring differences in concurrency control algorithms. Similarly, we are concerned with the data server queue length because we do not want transactions to spend the majority of their time waiting for disk access. Otherwise, we would be measuring disk response time rather than transaction response time. This measure also indicates if the work load is fairly distributed among the data servers. We are interested in differences in the average number of messages sent by each transaction because this also reflects a load on the system. We expect the passive schemes to perform better with this measure because conflict resolution requires only one message (from the cc node to the user node).

4.1.4. Baseline Simulations

Prior to experimenting with the concurrency control algorithms, a few preliminary simulations were conducted. The purpose was to establish reasonable values for parameters that would remain invariant for other experiments. Also, we wanted to check that simulations of the Ethernet and the data servers were correct. Basically, there were three questions to be answered. First, was the simulation of the Ethernet correct? Second, we had to determine a good ratio of user nodes to data servers. Too many transactions would cause the disks to become bottlenecks. Third, we wanted to determine transaction commit time without any concurrency control mechanism. These numbers would allow us to determine the amount of overhead added by a concurrency control algorithm.

Ether Utilization

To check the validity of the simulation of the Ethernet collision detection and retransmission protocol, we ran the Ethernet simulation under loads ranging from 10 to 90 percent of capacity. The packet size was fixed at either 256 bits or 2048 bits. In general, the Ether channel utilization matched the offered load up to about 85 percent of capacity. At higher loads, utilization leveled off but did not degrade. Also, allocation of channel capacity was fairly evenly divided among all hosts attempting to transmit, i.e. channel allocation was "fair". These results were consistent with previous studies (Shoch79a, Almes79a) of the Ethernet and led us to accept the simulation as correct.

Disk Utilization

A series of simulation experiments were conducted to determine the correct ratio of user nodes to data servers. In these runs, the number of data servers was fixed, the number of active transactions (i.e. user nodes) was varied and the average number of active (i.e. busy) disks was observed. Also, transactions constantly submitted requests (i.e. no thinktime time between requests). Thus, transactions spent all their time at the data servers and the difference between the number of active disks and the number of active transactions indicates the number of transactions in queues waiting for the disk. In Figure 28 we plot the number of active disks against the number of active transactions for 10 data servers. Also shown is a plot for 40 data servers. As can be seen, the number of active disks quickly levels off. This indicates that the number of data servers should exceed the number of user nodes. Otherwise, the response time would measure nothing but the queueing delay at the data servers and differences between concurrency control algorithms would be hard to distinguish.

It is also possible to apply an analytical result to the above problem. If the data servers are considered as cells and the transactions as balls, queueing transactions at data servers is the same as assigning balls to cells.

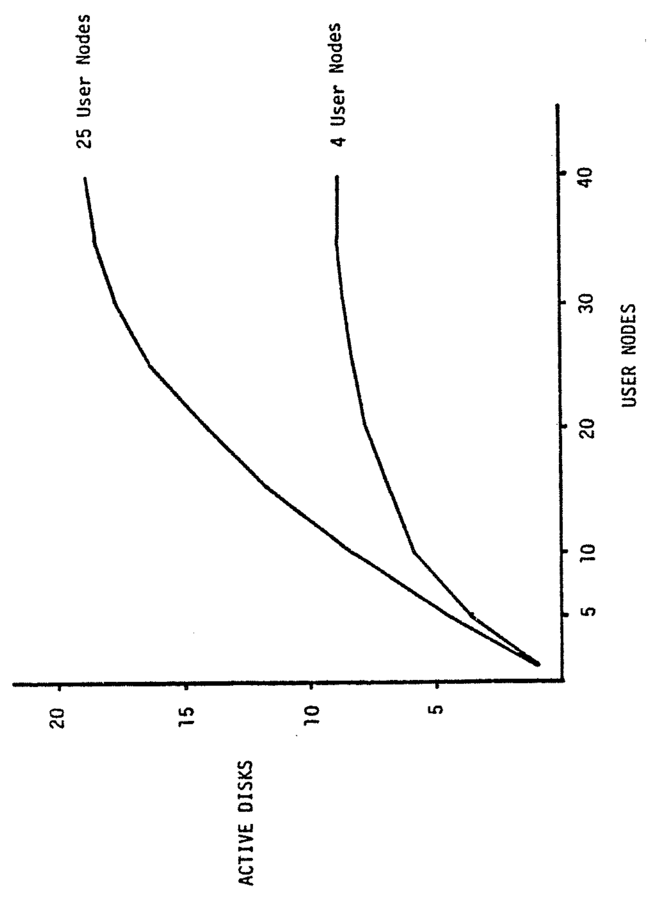

Figure 28. Disk Utilization vs. Number of Transactions

Given a random assignment, then, we are interested in the number of unoccupied cells, since this tells us the number of inactive disks. This problem reduces to an "occupancy" problem and can be approximated by a Poisson distribution [Feller68a]. The results of the approximation are also plotted in Figure 28. The close agreement between the approximation and the simulation is further evidence of the accuracy of the simulation.

Note that these numbers are pessimistic because they assume transactions are always queued at the data servers. This is not true in the concurrency control simulations so the result really gives an upper bound on the number of active disks.

Time in Commit

Several simulations were done of the entire system without any concurrency control mechanism. The intent was to establish lower bounds on commit time. In these simulations, the commit phase of transaction execution consisted only of the distributed two-phase commit protocol. No conflict analysis was involved. The difference between these "baselines" and the same measures of a system with concurrency control algorithm gives an indication of the overhead of the algorithm. Figure 29 plots commit time against message overhead for four and ten user nodes. The number

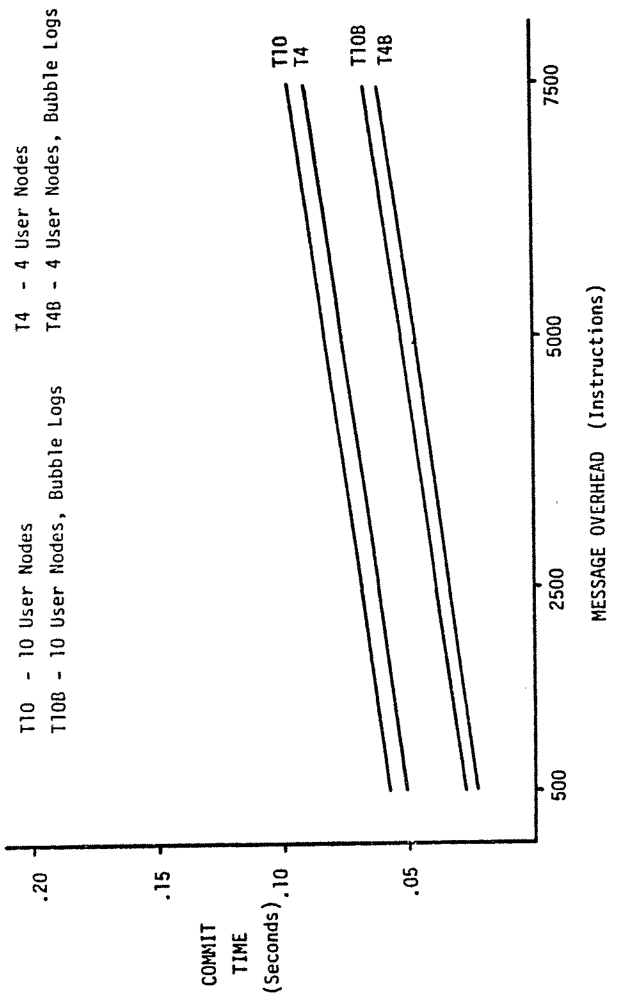

Figure 29. Baseline: Commit Time vs. Message Overhead

of data servers was fixed at 10. Note that commit time increases linearly with message overhead. This is reasonable since the time required to send the write and commit messages is increasing. The difference between 4 user nodes and 10 user nodes is constant. With 10 user nodes, transaction queueing time becomes a factor as indicated by the disk utilization plot. Only 6 disks are active under a load of 10 transactions so some transactions must be in disk queues. This difference between 4 and 10 user nodes reflects the time spent in disk queues. Also shown is commit time when the log files are maintained on bubble memories. Note that commit time drops by a large margin. This indicates that the time to write the log records on disk (at least 30 milliseconds) constitutes a large fraction of the commit overhead.

It may be argued that a system with so few user nodes is of little practical interest since the same performance (i.e. transaction throughput) can be had with a fairly powerful central computer. It would be possible to increase throughput by increasing the number of user nodes. However, as mentioned before, this would also increase the average queue length at the data servers. Our interest is in observing the effects of contention for data, not contention for disk access. But a more important point is that for measurements of commit time, the number of active transactions is relatively unimportant. The conflict rate parameter determines the frequency of data conflicts, not the number of transactions. For a constant conflict rate, the commit time should be invariant with respect to the number of user nodes (again, assuming disk contention is ignored).

Another point is that in a real system, the actual number of user nodes would be much higher. In our simulations, we made two simplifying assumptions which allowed us to simulate high database activity with a small number of user nodes. First, transactions were constantly queued at user nodes. A better model of reality would have been to schedule transaction "arrivals" at random intervals, perhaps according to a Poisson distribution. Second, the inter-request "thinktime" delay (Figure 22) was short in real-time. In an actual system, the inter-request delay might be longer to allow interaction with a user, output of a message to a terminal, etc. (although the number of machine instructions executed would stay the same). Both assumptions allowed us to simulate higher workloads with fewer numbers of users nodes. This decreased the size of the simulation and removed additional sources of variability (e.g. transaction arrival rate) that would have masked real differences among the concurrency control algorithms. Note that simulation experiments which measured throughput

included a larger number of user nodes.

4.2. Concurrency Control Experiments

Commit Time

The first set of experiments were designed to investigate differences in concurrency control overhead between the various algorithms. The experiments measured commit time for varying amounts of message overhead. Recall, that only transactions with conflicts were measured since transactions without conflicts incur very little overhead. There are three possible sources of waiting in the commit phase of execution: waiting for log records to be written by the data servers, waiting for locks to be released, and waiting for conflict analysis messages. Waiting for "ready" log records to be written by the data servers is required by the distributed two-phase commit protocol. Lock contention is a source of overhead only in the locking concurrency control schemes since transactions must wait for conflicting locks to be released before terminating. And conflict analysis messages are necessary only in the distributed schemes (to exchange portions of the dependency graph or send access restrictions). The relative weight of each of these components of commit time varies as explained below.

Figure 30 plots commit time for 10 data servers, 4 user nodes and a conflict rate of 0.1. Each transaction reads 3 records and updates 1 record. Note that with so few user nodes, there is very little queueing at the data servers (as can be seen on the disk utilization graphs). Requests are usually processed as soon as they arrive. At low message overhead, the non-locking algorithms have very similar response times. In this case, the cost of the two-phase commit protocol dominated the time in the commit phase. Because messages are so fast, all conflict analysis messages can be exchanged by the time the data servers write the log records. The locking algorithms are only slightly slower. This increase is due to lock contention. The passive locking scheme is faster than the distributed algorithm because the distributed algorithm requires at least two additional messages for conflict analysis. The lesson here is that if inter-process communication is very fast (perhaps through shared memory, for example), then the choice of a concurrency control algorithm is not too important. The principal source of delay in terminating transactions will be the distributed two-phase commit protocol.

As message overhead increases, commit time for the distributed algorithms increases at a faster rate than commit time for the passive algorithms. In fact, the passive locking scheme actually does better than the distributed

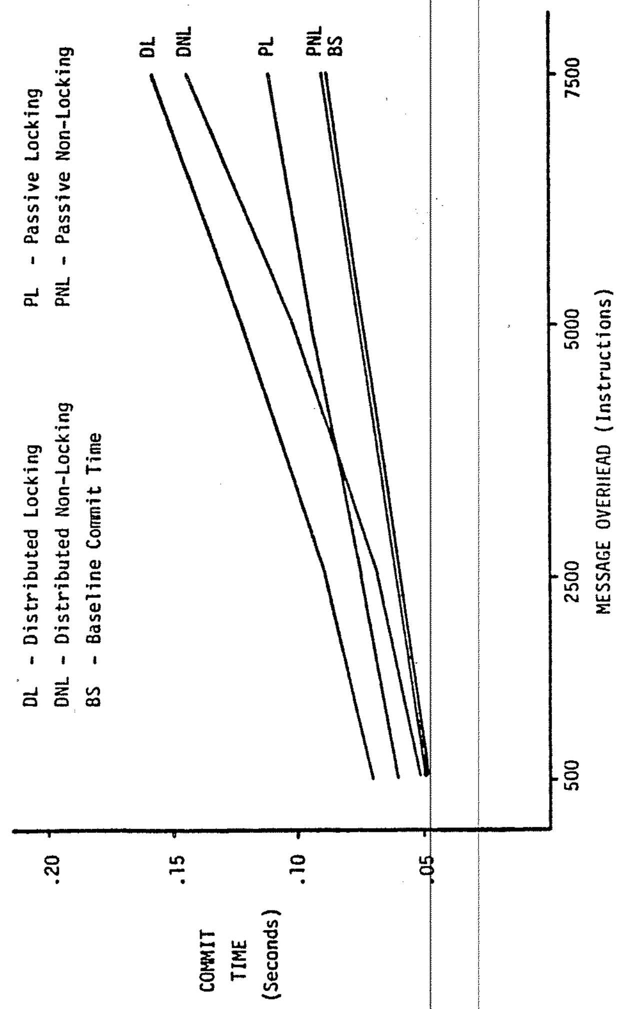

Figure 30. Commit Time vs. Message Overhead

non-locking algorithm when the message overhead exceeds approximately 3700 instructions. As mentioned above, the distributed algorithms require at least 2 more messages than the passive schemes to resolve conflicts. Thus, as message overhead increases, the cost of conflict analysis replaces the two-phase commit protocol as the dominant component of commit time.

We found it surprising to discover that the passive locking algorithm performed better than the distributed non-locking algorithm. We had expected that lock contention would cause significant delays in transaction commit time. This hypothesis was checked by examining the distribution of transaction commit times for the locking algorithms. It revealed that relatively few transactions had to wait for write locks for long periods of time. The majority of conflicts were resolved before the data servers had completed the log write. The histogram of commit times was essentially bimodal with a small peak due to transactions which had to wait for locks and a large peak where transactions were waiting for log writes to be completed. This may have been an artifact of the transaction size. Because transactions are short, locks were not held for long periods to begin with. However, a more important factor is the short thinktime delay between requests. In an actual system, this delay would be longer (in real-time)

which would effectively increase the time locks are held. The result would be larger differences between the locking and non-locking algorithms.

The results of the first set of experiments indicated that the passive algorithms outperform the distributed schemes. However, the differences were not as large as expected. One problem is that the time required to write log records at the data servers (about 30 milliseconds) is a substantial fraction of the time in the commit phase. This observation led to a second set of experiments in which we assumed that the log file was maintained on a bubble memory device. This reduced the time to write a log record by an order of magnitude (down to 3 milliseconds). The commit time for various values of message overhead is shown in Figure 31. For low message overhead, the non-locking algorithms benefited most from the use of bubble logs. Lock contention evidently kept the commit times high for the locking algorithms. However, as message overhead increases, the distributed algorithms perform nearly the same as without bubble logs. This phenomenon occurs quickly. In fact, the crossover point at which the passive locking scheme outperforms the distributed non-locking scheme is now about 2400 instructions (cf. 3700 instructions without bubble logs). This is evidence that conflict analysis dominates the cost of commit for the distributed

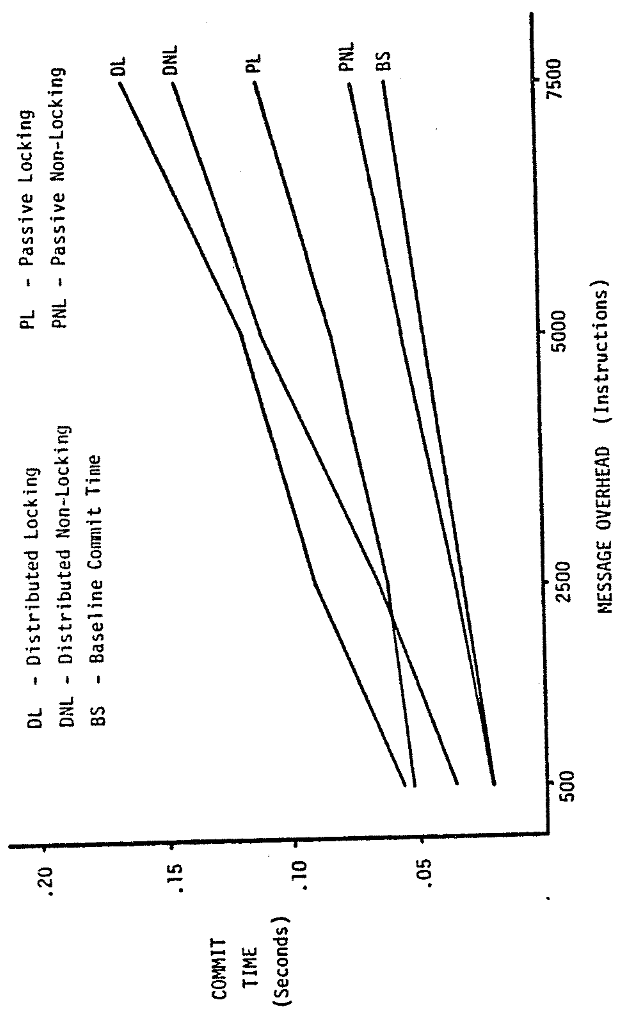

Figure 31. Commit Time vs. Message Overhead: Bubble Logs

algorithms. Note that commit time for the distributed locking algorithm is more than double the time for the passive non-locking algorithm.

It is interesting to compare the baseline commit times with the commit times for the passive non-locking algorithm. Recall that the baseline simulations involved no conflict analysis. The commit phase consisted only of the two-phase commit protocol. In Figure 30, the baseline and the passive non-locking line are nearly colinear. This says that the passive non-locking algorithm imposes almost no overhead on execution time. What is happening is that the cc node is able to complete conflict analysis long before the data servers complete the log writes. The effect of decreasing the time for log writes is apparent in Figure 31. There, the difference between the baseline and the passive non-locking algorithm increases with increasing message overhead. The reason for this is straightforward. In the baseline simulation, a user node must process 3 messages in order to commit a transaction: a write request, a ready log message, and a termination message. In the passive algorithm, a user node must process an additional message, the conflict analysis message from the cc node. When bubble logs are used, the conflict analysis time and the time to write the log records are more equal. So, both completion messages (from the cc node and the data servers) arrive at about the same time at the user node. Thus, the time to process the additional messages causes an increase in commit time. When bubble logs are not used, the commit time is dominated by the time to write log records. The time to process the conflict analysis message does not appear because the user node spends most of the commit time waiting on the data servers.

Throughput

Differences in commit time do not necessarily translate into differences in throughput. There are two reasons for this. First, commit time represents only a portion of total execution time (anywhere from 20 to 40 percent for a readset of 3 records). Thus, differences in commit time are diluted by time in the read phase. Second, commit time only varies for transactions with conflicts. Transactions without conflicts exhibit very similar commit times for all algorithms. Thus, for small conflict rates, overall differences in throughput may be negligible because so few transactions conflict. This is precisely the problem we encountered in attempting to get throughput measures. Typically, differences in throughput among the algorithms was at most 10 percent. The results from one set of experiments are shown in Figure 32.

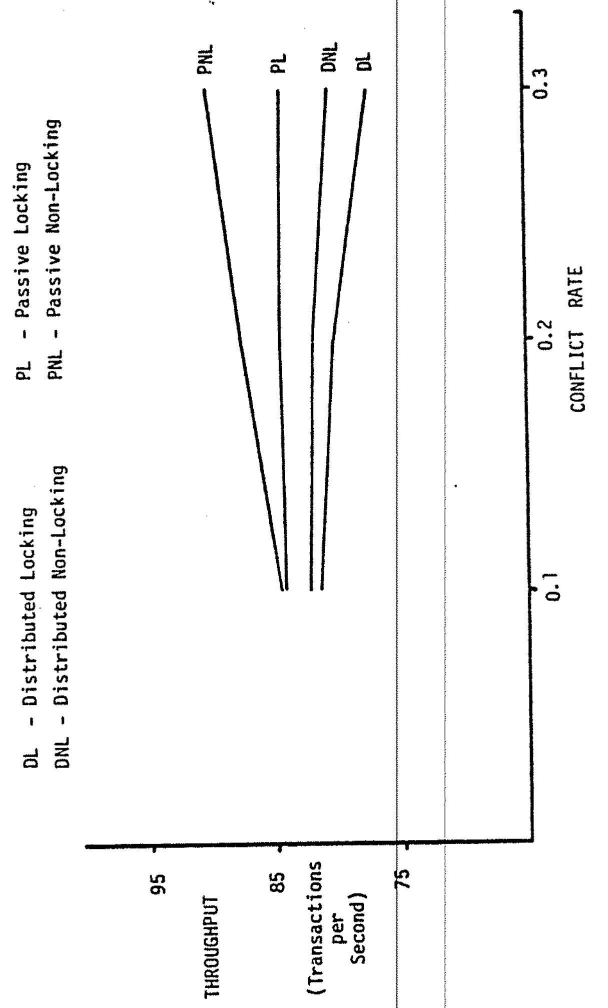

Figure 32. Throughput vs. Conflict Rate

These simulations were run with 25 user nodes and 40 data servers. This represents a more realistic load than that used above. The transaction size was the same as above (3 records read, 1 written). Bubble logs were used and the message overhead at the user nodes was fixed at 7500 instructions to maximize differences between the algorithms. Later we consider the effects of lower message overhead at the user nodes. Also, the capacity of the Ethernet was increased to 10 megabits per second. This was necessary because with so many concurrently active transactions, Ether contention had become a factor. Increasing the Ether capacity reduced Ether utilization from about 75 percent to 25 percent.

It is interesting to note that increasing the conflict rate can have two possible effects. First, a higher conflict rate may simply increase the number of transactions with conflicts. Second, increasing the conflict rate could cause a transaction to interfere with increasing numbers of other transactions, i.e. a transaction may conflict with two or three other transactions, rather than just one. In these simulations, because transactions are small, we expected that a transaction will conflict with at most one other transaction. Thus, increasing the conflict rate should only increase the total number of transactions with conflicts. It should not increase the commit time, however. In these runs, then, a conflict rate of 0.3 implies approximately 30 percent of the transactions required conflict analysis.

There are several noteworthy points in Figure 32. First, the throughput for the centralized non-locking scheme actually increases with higher conflict rates. This seems counter-intuitive since more conflicts should require more work. However, recall that the data servers maintain a data cache and requests are served from buffers whenever possible. Increasing the conflict rate, then increases the number of hits in the cache and so reduces the amount of total I/O. The passive non-locking scheme is able to take advantage of this because of its very low overhead. The distributed algorithms could not because the cost of a conflict message is so high. Note that all algorithms have similar throughput for low conflict rates. This is true even though commit time differences are large for transactions with conflicts (see Figure 31). This reinforces our claim that the concurrency control algorithm has little effect on overall throughput for low conflict rates.

A second point is that the performance difference between the passive non-locking and the distributed locking algorithms is only 12 and 16 percent at conflict rates of 0.2 and 0.3, respectively. This is not a large difference. Further investigation showed the difference was due solely to the cost of conflict analysis messages in the distributed algorithms.

A third point worth mentioning is that the performance degradation for the distributed algorithms is non-linear. This is surprising. We expected that higher conflict rates would simply increase the total number of transactions with conflicts (and so cause a linear change in performance). Contrary to our conjecture above, it seems that the higher conflict rate did lead to increased conflict analysis time. To check this, we examined the number of messages sent and received by transactions with conflicts in the distributed non-locking algorithm. We found the average number of messages processed increased from 12.45 to 13.46 for conflict rates of 0.1 and 0.3, respectively. Since transaction size is constant, the increase can only be due to more conflict analysis messages being exchanged. Because messages are expensive, this shows up as a performance loss.

For contrast, we ran the same experiment as above with a message overhead of only 2500 instructions at the user nodes. We expected that the decrease in communication cost would benefit the distributed concurrency control algorithms. The results are shown in Figure 33. Note that the scale on the throughput axis has increased by approximately 40 transactions per second. Thus, the decrease in message overhead has a much more dramatic effect on throughput than

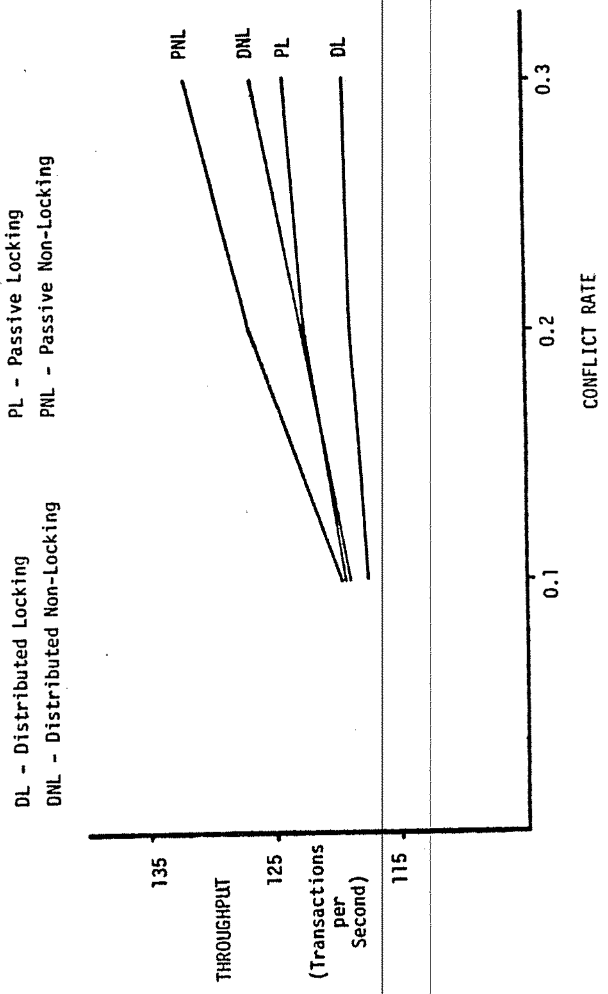

Figure 33. Throughput vs. Conflict Rate: Low Overhead

differences among the concurrency control algorithms. Also note that throughput for all schemes now increases with the higher conflict rate. The message overhead is low enough that the effect of caching in the data servers is evident. The distributed non-locking algorithm now performs as well or better than the passive locking algorithm. Interprocessor communication is faster than waiting for locks to be released. Finally, note that the passive non-locking scheme still out-performs the other algorithms. But the performance increase is not as large as before; at a conflict rate of 0.3, the differences between the passive non-locking algorithm and the the distributed non-locking and locking schemes are only 4 and 10 percent, respectively.

4.3. Discussion

In this section, we discuss and criticize the results of the simulation. One important result is that the cost of the distributed two-phase commit protocol is very high. Not only does it increase the work load at the data servers but it also slows transaction commit because transactions must wait for acknowledgments that the logs have been written. A corollary of this result is that when the message overhead is not high, the choice of a concurrency control algorithm is immaterial because essentially all the conflict analysis can be completed by the time the data

servers have completed their log writes. This suggests that reducing the cost of maintaining the log file is an area to be investigated.

A possible solution is to keep the log on a bubble memory as has been simulated here. Other technologies, such as magnetic core storage, are available as well. However, the general problem of guaranteeing that all sites unanimously commit or abort the transaction seems to be inherently complex. The difficulty is that crash recovery is done autonomously at each site so there must be complete agreement among the sites before committing the transaction. An alternative, mentioned previously, is to give one site sole responsibility for recovery for the entire system. During crash recovery, a processor would consult the recovery node for database restoration. This would greatly simplify and speed up the recovery protocol (just as centralization simplifies concurrency control). The distributed two-phase commit protocol would not be necessary. Only an acknowledgment from the recovery node would be needed for commit. Centralized recovery seems feasible for business applications since transactions update only a small amount of data. It is an area for further research.

Another result of interest is that under low conflict rates (i.e. 0.1), the performance of all algorithms is similar (at most a 4 percent difference). Again, this suggests that any concurrency control algorithm is acceptable. It is difficult to know what a "reasonable" conflict rate is, however. It is really a function of the application and the transaction size and locking granularity.

Unlike other simulations of centralized concurrency control algorithms, our passive schemes do not exhibit sharp performance drops under a heavy load. This may be partly due to differing assumptions about the cost of locking, message handling, etc. However, it was difficult to construct a scenario in which the cc node became a bottleneck. Simply increasing the transaction load did not work. We attempted to increase the load 3 ways: lowering message overhead, raising the number of user nodes, and making user nodes as fast as the cc node and data servers. Unfortunately, all attempts to increase the load also increased the load at the data servers. Even though the load increased at the cc node, the delay caused by queues at the data servers more than offset the cc node delay. The passive non-locking scheme invariably had the highest throughput (even with a message overhead of only 2500 instructions).

We did succeed in causing the cc node to become a bottleneck. However, it was done by increasing the message overhead at the cc node, not by increasing the work load. In the previous experiments, the message processing

overhead at the cc node and the data servers was fixed at 1500 instructions. But here, we varied the message overhead at the cc node and data servers and measured the throughput. The conflict rate was fixed at 0.2 and the user node message overhead was fixed at 7500 instructions. The results appear in Figure 34. Throughput for the distributed algorithms decreases somewhat due to the increased message overhead at the data servers. However, the effect of higher message overhead on the cc node is dramatic. From an overhead of 1500 instructions to an overhead of 7500 instructions, throughput drops by 40 percent for the passive algorithms. We also examined the average number of outstanding request messages in the input queue of the cc node. At an overhead of 2500 instructions, there is an average of one message in the input queue. This represents very little delay. However, at an overhead of 7500 instructions, the number of requests in the input queue exceeds 20. Thus, when a transaction broadcasts a write request, the user node must wait for the cc node to analyze at least 20 other request messages before it can respond to the commit request. This causes a large increase in commit time for all transactions which is evident by the decreased throughput. These results emphasize the necessity of low message processing overhead at the cc node if high rates of transaction throughput are to be maintained.

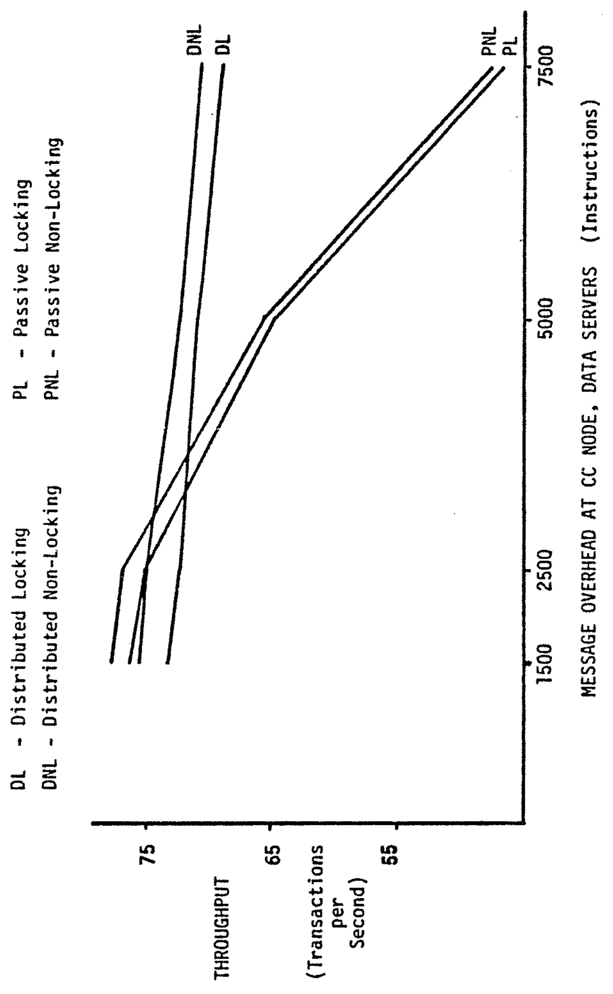

Figure 34. Throughput vs. CC Node, Data Server Message Overhead

We feel the transaction size used in our experiments is reasonable for a business application. It is interesting that when such transactions conflict, the conflict is usually between two transactions only. Most deadlock cycles are also of length two. Thus, sophisticated deadlock detection algorithms are really wasted in such environments. Also, the distributed algorithms tended to process 2-3 more messages than the passive algorithms in order to do conflict analysis. Therefore, if one is designing a log protocol for a database system using distributed conflict analysis, the log operation should be completed within about 3 message times.

One problem with the simulations is that not all costs were incorporated. For example, suppose B-trees are used to index data records. Then nodes of the B-tree must be locked as well as data records. If a data record is moved or its index value is modified, the B-tree is another source of contention which could affect performance.

Another problem with the experiments is that differences between the locking and non-locking algorithms were distorted by the small transaction size and the small inter-request interval. For example, we were surprised to find the passive locking algorithm perform better than the distributed non-locking algorithm under any circumstances. In an actual system, we would expect locks to be held for longer intervals (during user interaction, schema lookup, etc.) which would increase waiting time. Thus, real differences between the locking and non-locking algorithms are probably greater than indicated here. However, our results may be interpreted as lower bounds on the performance differences.

CHAPTER 5

SUMMARY

5.1. Conclusions

The major contributions of this research are twofold. First, we have proposed a novel architecture for business database applications and designed a very low overhead concurrency control algorithm for the system. Second, we have shown that the cost of recovery protocols (in particular, the distributed two-phase commit protocol) can outweigh the overhead of a concurrency control algorithm.

We feel that the architecture we have presented is a viable approach for a high throughput, transaction-oriented system. Separation of transaction management from data access allows a high degree of parallelism among independent tasks. The Ethernet provides a high-speed interprocessor communications medium. The use of personal computers for user nodes provides a cost savings since high performance processors are unnecessary for a user interface. Data servers permit concurrent access to high cost peripherals (i.e. disks). Given this architecture, we have shown how concurrency control may be performed in parallel with database accesses. The passive technique imposes almost no overhead on the system. Thus, it allows the highest throughput of any algorithm. The technique can be made robust with respect to lost messages and failed processors at no performance penalty. We point out that to do this we developed an acknowledgment protocol for broadcast messages. This is important since termination messages and write requests should be broadcast whenever possible. To our knowledge, such an acknowledgment protocol has not been published before.

We feel that our work vindicates centralized concurrency control algorithms. This is significant since centralized schemes are relatively simple and easy to understand. The performance analysis showed that the passive scheme performed better than the distributed schemes under conditions which would normally favor a distributed algorithm (i.e. resources not pre-claimed). By taking advantage of the broadcast nature of the Ethernet, we were able to eliminate explicit lock request and lock grant messages which had plagued earlier centralized schemes. Finally, so long as the message processing overhead and the conflict analysis overhead at the central node is not high, the probability that the central node will become a system bottleneck can be kept at a minimum. It does not appear that the overhead of conflict analysis (constructing the global dependency graph, etc.) by itself is enough to create a bottleneck.

Previous performance analyses of concurrency control algorithms failed to include the cost of recovery protocols. Under certain assumptions, this cost can be significant. We have assumed an "optimistic" approach to concurrency control, i.e. updates are not declared until the transaction wishes to terminate. At this point, conflict analysis is performed and if the updates are acceptable, the transaction may commit. We observed that the distributed two-phase commit protocol (used for recovery) may proceed in parallel with conflict analysis. And, in most cases, conflict analysis can be completed before the two-phase commit protocol completes. The problem is that the recovery protocol requires log records to be written. The major cost of concurrency control is message passing. Thus, if the message overhead is low, the choice of a concurrency control algorithm is unimportant. The real impediment to throughput is the recovery protocol.

This suggests that a factor to consider in designing a database system is the relative cost of message passing versus the time to write log records. A low ratio implies that the transaction will spend much more time waiting for log records than performing conflict analysis, regardless of the concurrency control algorithm. Note that the distributed two-phase commit protocol is not the only recovery algorithm. Non-blocking commit protocols are described in [Skeen81a] but they require even more messages than the basic two-phase protocol. Protocols requiring fewer messages than the two-phase commit are described in [Gray80a]. However, he assumes that transactions migrate from one processing site to another and never return to a site. This is not true in our model. Other possibilities are to reduce the cost of the two-phase commit by reducing the time to write log records. When bubble memories were simulated as the log medium, differences between the concurrency control algorithms became apparent.

As a general comment on research in concurrency control, it is worth pointing out that most transaction conflicts were quickly resolved. At the highest conflict rates, transactions with conflicts in the distributed schemes processed on the average only 2.5 more messages than conflicting transactions in the passive schemes. This suggests that conflicts are indeed rare and that the overhead of dealing with them is not high.

5.2. Future Work

The results of this research pose several interesting questions and suggest areas for future investigation. An immediate question is how might other database management functions be incorporated into the architecture. For example, should there be a dedicated node to provide indexing

information or should user nodes perform their own indexing? We are also interested in the performance effects of other database functions. When recovery was included in the simulation model, it significantly affected the results. Our ultimate goal is to design a system which can provide a throughput of 100 transactions per second. The results for throughput in Chapter 4 exceed this rate. It is not clear to what extent the additional costs will reduce this performance.

As mentioned above, an efficient recovery protocol would be very valuable. This could mean a better implementation of the distributed two-phase commit protocol or an entirely new protocol. One possibility is investigating the passive recovery scheme with batched log records as outlined in Section 3.5.

One aspect of the simulations with which we are uncomfortable is the values of message processing overhead. We claimed that the cc node and data servers can process a message in 1500 instructions. However, we have no way of proving this claim. Thus, it would be interesting to continue along the lines of [Specto80a] and find efficient processing techniques for other message classes besides datagrams. This would tell us what the minimum amount of message overhead is in the cc node and thus the maximum transaction processing rate.

One difficulty we encountered in this research was the lack of flexible, easy-to-use tools for performance analysis. The simulations were especially difficult, although the use of SIMPAS [Bryant80a] was a great help and saved many months of effort. The problem is that we were simulating a distributed system with a sequential programming language (PASCAL). For example, it was difficult to represent sending a request and waiting a reply. All context information is lost once the send procedure is exited. The result was that such constructs were difficult to code, understand, and modify. What we desire is a distributed programming language which can be used as the basis for a software testbed. For example, we might want to emulate portions of a distributed system but simulate others. Also, we might like to be able to alter the underlying message subsystem without any changes to applications which use it. A first step in this regard is *MOD [Cook79a]. It would be interesting to incorporate the notion of time into such a language. For example, one could keep a local clock for each simulated processor and advance the clock for each statement executed. This would allow one to simulate different processor speeds or to easily switch from emulation to simulation.

One final point is that an underlying assumption in this research is that, for business applications, record-

at-a-time access is more efficient than associative search. However, given some recent work in associative disk designs, that may no longer be the case [Boral81a, Bancil80a]. For example, a B-tree search may require four or five disk accesses for a very large index. It may be possible to simply index down to the cylinder level and associatively search the entire cylinder in less time than it takes to search the B-tree. If that is the case, our passive scheme becomes more complex. This is because data servers would presumably re-order requests so that two outstanding requests for the same cylinder could be processed in the same revolution. Recall that the cc node requires that requests be processed in the order sent over the Ether so that its conflict information is consistent. With this new technology in mind, we plan to study the relative cost of indexing versus associative search in a business database sometime in the future.

REFERENCES

[Almes79a]G.T. Almes and E.D. Lazowska, "The Behavior of Ethernet-Like Computer Communications Networks," Proc. Sigops Conference, (1979).

[Alsber76a]P.A. Alsberg, "A Principle of Resilient Sharing of Distributed Resources," Proc. 2nd Int'l Conf. on Software Engineering, (October 1976).

[Badal79a]D.Z. Badal, "Correctness of Concurrency Control and Implications in Distributed Databases," Proc. IEEE COMPSAC, (November 1979).

[Badal80a]D.Z. Badal, "The Analysis of the Effects of Concurrency Control on Distributed Database System Performance," Proc. VLDB-6, (October 1980).

[Bancil80a]F. Bancilhon and M. Scholl, "Design of a Backend Processor for a Data Base Machine," Proc. of the ACM SIGMOD 1980 Int'l Conf. on Management of Data, (May 1980).

[Banerj79a]J. Banerjee, D.K. Hsiao, and K. Kannan, "DBC - A Database Computer for Very Large Databases," IEEE Transactions on Computers c-28, 6, (June 1979).

[Banino79a]J.S. Banino, C. Kaiser, and H. Zimmermann, "Synchronization for Distributed Systems Using a Single Broadcast Channel," Proc. of the 1st Int'l Conf. on Distributed Computing Systems, (October 1979).

[Belfor79a]G.G. Belford and E. Grapa, in a Distributed Computing System"" "Setting Clocks "Back" in a Distributed Computing System," Proc. of the 1st Int'l Conf. on Distributed Computing Systems, (October 1979).

[Bernst81a]P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys 13, 2, (June 1981).

[Bernst78a]P.A. Bernstein, J.B. Rothnie, N. Goodman, and C.H. Papadimitriou, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases," IEEE Transactions on Software Engineering 4, 3, (May 1978).

[Bjork73a]L. Bjork, "Recovery Scenario for a DB/DC System," Proc. ACM National Conference, (1973).

[Boral80a]H. Boral, D.J. DeWitt, D. Friedland, and W.K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," Computer Sciences Department Technical Report No. 402, University of Wisconsin (October 1980).

[Boral81a]H. Boral, D.J. DeWitt, and W.K. Wilkinson, "Performance Evaluation of Four Associative Disk Designs," Information Systems, (Submitted March 1981).

[Bryant80a]R. M. Bryant, "SIMPAS User Manual," Computer Sciences Technical Report #391, University of Wisconsin - Madison (June 1980).

[Cheng80a]W.K. Cheng and G.G. Belford, "A Clock Synchronization Algorithm for Update Synchronization on Local Broadcast Networks," Proc. Fall 1980 Compcon, (1980).

[Cook79a]R.P. Cook, "*MOD - A Language for Distributed Programming," Proc. of the 1st Int'l Conf. on Distributed Computing Systems, (October 1979).

[Dadam80a]P. Dadam and G. Schlageter, "Recovery in Distributed Databases Based on Non-synchronized Local Checkpoints," IFIP Congress, (1980).

[Dalal78a]Y.K. Dalal and R.M. Metcalfe, "Reverse Path Forwarding of Broadcast Packets," CACM 21, 12, (December 1978).

[Davies73a]C.T. Davies, "Recovery Semantics for a DB/DC System," Proc. ACM National Conference, (1973).

[DeWitt79a]D.J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers c-28, 6, (June 1979).

[DeWitt81a]D.J. DeWitt, Personal Communication. (June, 1981).

[Eswara80a]K.P. Eswaran, Personal Communication. (April 1980).

[Eswara76a]K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," CACM 19, 11, (November 1976).

[Farber73a]D.J. Farber, J. Feldman, F.R. Heinrich, M.D. Hopwood, K.C. Larson, D.C. Loomis, and L.A. Rowe, "The Distributed Computing System," Proc. 7th Annual IEEE Computer Society International Conference, pp. 31-34 (February 1973).

[Feller68a]W. Feller, An Introduction to Probability Theory and Its Applications, J. Wiley & Sons (1968) 3rd Edition.

[Garcia79a]H. Garcia-Molina, "Centralized Control Update Algorithms for Fully Redundant Distributed Databases," Proc. of the 1st Int'l Conf. on Distributed Computing Systems, (October 1979).

[Garcia79b]H. Garcia-Molina, Performance of Update Algorithms for Replicated Data in a Distributed Database, Computer Sciences Department, Stanford University (June 1979) Ph.D. Thesis.

[Gelenb78a]E. Gelenbe and K. Sevcik, "Analysis of Update Synchronization for Multiple Copy Data Bases," Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, (August 1978).

[Gray78a]J.N. Gray, "Notes on Database Operating Systems," Report RJ2188, IBM, San Jose, California (1978).

[Gray80a]J.N. Gray, "Minimizing the Number of Messages in Commit Protocols," Workshop on Fundamental Issues in Distributed Computing, (December 1980).

[Gray81a]J.N. Gray, P. Homan, H. Korth, and R. Obermarck, "A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System," Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks, (February 1981).

[Gray76a]J.N. Gray, R.A. Lorie, G.F. Putzolu, and I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database," in Modeling in Data Base Management Systems, ed. G.M. Nijssen,North Holland (1976).

[Gray81b]J.N. Gray, P. McJones, M. Blasgen, B.G. Lindsay, R.A. Lorie, T. Price, G.F. Putzolu, and I.L. Traiger, "The Recovery Manager of the System R Database Manager," ACM Computing Surveys 13, 2, (June 1981).

[Hawtho81a]P. Hawthorn, "The Effect of the Target Applications on the Design of Database Machines," Proc. of the ACM SIGMOD 1981 Int'l Conf. on Management of Data, (May 1981).

[Hsiao81a]D.K. Hsiao and T.M. Ozsu, "A Survey of Concurrency Control Mechanisms for Centralized and Distributed Databases," Technical Report OSU-CISRC-TR-81-1, Ohio State University (February 1981).

[IBM80a]Corporation IBM, "IBM 3380 Direct Access Storage Description and User's Guide," IBM Document GA26-1664-0, File No. S/370-07,4300-07 (1980).

[Kaneko79a]A. Kaneko, Y. Nishihara, K. Tsuruoka, and M. Hattori, "Logical Clock Synchronization Method for Duplicated Database Control," Proc. of the 1st Int'l Conf. on Distributed Computing Systems, (October 1979).

[Kung81a]H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," ACM TODS 6, 2, (June 1981).

[Lampor78a]L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," CACM 21, 7, pp. 558-565 (July 1978).

[Lampso79a]B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," Submitted to CACM, Computer Science Laboratory, Xerox PARC, (1979).

[LeLann78a]G. LeLann, "Algorithms for Distributed Datasharing which use Tickets," Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, (August 1978).

[Lomet80a]D.B. Lomet, "Deadlock Avoidance in Distributed Systems," IEEE Transactions on Software Engineering 6, 3, (March 1980).

[Menasc80b]D.A. Menasce and T. Nakanishi, "Optimistic Versus Pessimistic Concurrency Control Mechanisms in Database Management Systems," Technical Report, Pontificia Universidade Catolica do, Rio de Janeiro (1980).

[Menasc80a]D.A. Menasce, G.J. Popek, and R.R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases," ACM TODS 5, 2, (June 1980).

[Metcal76a]R.M. Metcalfe and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM 19, 7, pp. 395-403 (July 1976).

[Minker78a]J. Minker, Logic and Data Bases, Plenum Press, New York (1978).

[Montgo79a]W.A. Montgomery, "Polyvalues: A Tool for Implementing Atomic Updates to Distributed Data," Proc. ACM Symposium on Operating Systems, (1979).

[Ozkara75a]E.A. Ozkarahan, S.A. Schuster, and K.C. Smith, "RAP - An Associative Processor for Data Base Management," Proc. NCC 45, AFIPS Press, (1975).

[Papadi79a]C.H. Papadimitriou, "The Serializability of Concurrent Updates," JACM 26, 4, (October 1979).

[Reed78a]D.P. Reed, "Naming and Synchronization in a Decentralized Computer System," Technical Report MIT/LCS/TR-205, MIT (September 1978).

[Ries79a]D.R. Ries, The Effects of Concurrency Control on the Performance of a Distributed Management System, Computer Sciences Department, University of California, Berkeley (March 1979) Ph.D. Thesis.

[Ries78a]D.R. Ries and M.R. Stonebraker, "Locking Granularity Revisited," ACM TODS 4, 2, (June 1978).

[Rosenk78a]D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, "System Level Concurrency Control for Distributed Database Systems," ACM TODS 3, 2, (June 1978).

[Shoch79a]J.F. Shoch and J.A. Hupp, "Performance of an Ethernet Local Network: A Preliminary Report," Proc. of the Local Area Communications Network Symposium, (May 1979).

[Skeen81a]D. Skeen, "Nonblocking Commit Protocols," Proc. of the ACM SIGMOD 1981 Int'l Conf. on Management of Data, (May 1981).

[Specto80a]A.Z. Spector, "Performing Remote Operations Efficiently on a Local Network," Stanford Technical Report STAN-CS-80-831, Stanford University, (December 1980).

[Stoneb78a]M.R. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres," Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, (August 1978).

[Stoneb76a]M.R. Stonebraker and E. Neuhold, "A Distributed Data Base Version of INGRES," Memorandum No. ERL-M612, Engineering Research Laboratory, University of California - Berkeley (September 1976).

[Tanenb81a]A.S. Tanenbaum, Computer Networks, Prentice-Hall, Englewood Cliffs, N.J. (1981).

[Teraji80a]M. Terajima, "Recent Progress in Memory Devices and Their Prospects," IFIP Congress, (1980).

[Thomas79a]R.H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM TODS 4, 2, (June 1979).

[Tokoru77a]M. Tokoru and K. Tamaru, "Acknowledging Ethernet," Proc. Fall 1977 Compcon, (September 1977).

[Yemini79a]Y. Yemini and D. Cohen, "Some Issues in Distributed Processes Communication," Proc. of the 1st Int'l Conf. on Distributed Computing Systems, (October 1979).