DATABASE CONCURRENCY CONTROL IN LOCAL BROADCAST NETWORKS

by

David J. DeWitt
and
W. Kevin Wilkinson

Computer Sciences Technical Report #396

August 1980

Abstract

This paper proposes a scheme for "passive" concurrency control to be implemented on a local broadcast network (e.g. Ethernet). It is termed passive because no explicit synchronization messages are required. Conflict analysis is done on-the-fly by a single node which eavesdrops for database requests on the broadcast medium. This approach is feasible for applications characterized by a high volume of small, update-intensive transactions. We believe this scheme can have better performance (fewer messages, fewer transaction aborts, higher throughput) than previously proposed distributed concurrency control algorithms.

1. Introduction

In this paper, we propose a passive concurrency control scheme for use in a local, distributed database system or a database machine with a broadcast capability. It is termed passive because no explicit synchronization messages are required. A dedicated node performs conflict analysis by eavesdropping on the "ether" for read and write requests and actively controlling the commit process. The first part of this paper discusses the application environment for which this concurrency control mechanism is intended. Next, we briefly review some existing concurrency control algorithms and discuss performance problems which might arise in an implementation of those schemes in our environment. Then we present the passive concurrency control scheme. Finally, we mention some problems with the scheme and areas for future research.

Our recent work on database machines has concentrated on improving the performance of very data-intensive applications through the use of tightly-coupled multiprocessor systems with massive parallelism and centralized control [DeWitt79, Boral8Øa, Boral8Øb]. Such machines, however, do not appear to be the most cost effective solution for those applications which are characterized by a high volume of small transactions. We classify a transaction as "small" if it reads and writes only a few entities corresponding to records or tuples (as opposed to scanning whole relations). The set of transactions would be less retrieval oriented and more update intensive. Examples of such applications are banking systems or airline reservation systems. In a

banking system, a large number of the transactions would simply credit or debit a single account. Clearly, large transactions would occur (such as monthly summary statements) but we assume those are so infrequent as to constitute a negligible portion of the transactions run.

System D [Eswaran8Ø] appears to be the first multiprocessor organization designed specifically for handling this type of application. Our work in this area has certainly been motivated by the goals of System D in addition to the recognition that database machines such as DIRECT [DeWitt79] are not the best organizations for this type of application. The organization of our system is described briefly in Section 4. We feel that both System D and our system can be viewed as either a local, distributed database systems or as a loosely-coupled database machine.

Much of the work in distributed database systems has assumed an ARPANET framework, i.e. the underlying network is widely geographically dispersed, connecting large, general-purpose machines. Inter-processor messages may be forwarded from site to site and take a relatively long time for delivery. An alternative architecture for distributed databases is a local broadcast network, best exemplified by Ethernet [Metcalfe76]. Such networks have relatively high-speed, high bandwidth inter-processor communication. In addition, they provide distributed control for access to the common communications medium and the communications medium (the "ether") is passive. These features increase the reliability of the network and the failure of a single site does not affect the rest of the network. The Ethernet has been shown

to be stable at high utilization rates [Shoch80]. Thus, throughput will not sharply decline during periods of high activity on the network (in the case of bursty traffic). We feel that for the applications we have in mind, a local broadcast network is a more suitable architecture than a geographically dispersed, site-to-site transmission network. The high volume of transactions places a premium on fast communications. The failure of a site on the ARPANET could significantly increase the cost of communication if the failed node provides a critical link between two distant processors. This is not true on the Ethernet. Here, the cost of messages is independent of the reliability of individual sites. Also, transactions are small, with modest computing requirements that could easily be handled by smaller machines attached to the broadcast medium. On a site-to-site network, the computations might be performed faster, but since transactions are short, most of the time the transaction will be waiting on messages. The ratio of computing and message requirements in our environment suggests that an Ethernet might provide a better balance between the two than an ARPANET.

2. Existing Solutions to Concurrency Control

There have been many proposals for concurrency controls in distributed systems. The following list is representative and not exhaustive. The centralized concurrency control algorithms have a dedicated processor handling conflict analysis. In one scenario, this site controls all accesses and resource requests are directed to it [Menasce80a]. In another scenario, local con-

currency controllers monitor requests at individual sites. The central site is only notified when the local controllers detect a conflict involving non-local transactions [Stonebraker76], [Gray78]. The Wound-Wait-Die concurrency control algorithms [Rosenkrantz78] use transaction timestamps (globally unique) to resolve conflicts. Their model assumes that transactions migrate from site to site requesting resources from a local concurrency controller at each site visited. Global deadlock and infinite restart are avoided by having all the local concurrency controllers resolve conflicts in a uniform way based on the timestamps of the transactions involved.

The Majority Consensus algorithm proposed by [Thomas79] combines a concurrency control algorithm with a scheme for keeping copies of data consistent at all sites where they exist. Here, data items are timestamped rather than transactions. When a transactions begins, it first obtains the timestamps of all entities it will access during execution (called the base set). During commit processing, those timestamps are compared with the current entity timestamps. If any have changed, an update has occurred during the life of the transaction. Therefore, the base set is now obsolete and the transaction must abort and restart.

The SDD-1 approach to concurrency control [Rothnie79] involves the notion of transaction classes. A transaction class is defined by a set of entities read and a set of entities written (i.e. a logical subset of the database). The classes are specified at database design time by the database administrator. When a transaction begins, its resource requirements are pre-

specified so it may be placed in a transaction class. Then, conflict analysis is simplified since transactions conflict only if their respective classes conflict and class conflicts are predetermined at database design time. The concurrency controller, since it knows the class of each transaction and how the classes conflict, knows which transactions have to synchronize against each other and how to synchronize them. Several conflict resolution strategies exist based on the nature of the conflict between transaction classes.

System D is a local, distributed database management system under development at IBM. Unlike the other systems, System D was designed with a local broadcast network in mind. The system is composed of two types of processing sites: transaction nodes and data nodes. Transaction nodes provide a user interface and monitor transaction execution. Data nodes manage a portion of the database and perform concurrency control. Transaction nodes submit read and write requests to data nodes which process the request. A read request returns the current value of an entity. Write requests are applied to a temporary workspace for the transaction until the transaction commits, when the writes are applied to the database. The idea behind concurrency control in System D may be motivated with the following example. Suppose we have two transactions, T1, T2 and the request sequence:

```
T1: read  x
T2: read  y
T2: write x
T2: commit
```

The correctness condition for concurrency controllers is that the result of interleaving transactions be equivalent to executing those transactions serially [Eswaran76]. In the above example, the effective execution order is T1, T2. The problem is that transaction T1 is still active. If it subsequently makes a request to write y, we get a non-serializable request sequence (since, if T1 really executed first, T2 would have read a different y value). System D would prevent this by prohibiting T1 from writing anything read by T2. Therefore, when a transaction commits, it may impose access constraints on other active transactions. These access constraints are the essence of System D concurrency control. In a distributed system, these constraints must be broadcast to all data nodes involved in the processing of the conflicting transactions. A single request may trigger a flurry of messages as data nodes communicate to update their access constraints. When a non-serializable request sequence occurs, the offending transaction is aborted. The advantage to this scheme is that requests are never delayed for concurrency control purposes (there are no locks). The disadvantage is the increased message activity when conflicts occur. But, the designers of System D are aiming at the same application environments we have described above (i.e. banking systems, airline reservations). So, transactions are short and when conflicts occur the number of "update constraint" messages would be limited. The broadcast capability of the network is what makes this algorithm feasible since all relevant data nodes can be notified of a conflict with the same message. On a site-to-site

transmission network, individual messages would need to be sent. Thus, System D would probably not work well in an ARPANET environment.

3. Problems with Existing Solutions on Local Broadcast Networks

It is possible to distinguish two different philosophies of concurrency control which might be termed optimistic and pessimistic [Menasce80b]. The optimistic philosophy is that conflicts practically never occur. When they do occur, relatively simple conflict resolution strategies may be used to resolve them. These strategies may not be the most efficient, but since conflicts are rare, the impact on performance is negligible. Concurrency control algorithms which seem to embrace this philosophy are Wound-Wait-Die and to some extent Majority Consensus. For example, the Wound-Wait-Die scheme is conservative and aborts transactions unnecessarily. The Majority Consensus algorithm will also abort transactions unnecessarily. And when an abort occurs, it is during the commit processing. All the work done to execute the transaction is wasted. The designers of both schemes have suggested some changes to improve performance but, of course, they increase the cost.

The pessimistic philosophy of concurrency control is that conflicts are not rare. SDD-1, System D, and centralized schemes are examples. These schemes use much more efficient conflict resolution strategies. Transactions are never aborted for concurrency control reasons in SDD-1 (but they may be delayed).

System D aborts transactions only when necessary and does so as soon as the request causing the conflict is processed (unlike Majority Consensus, no extra work is wasted). The cost of more efficient conflict resolution is more information in SDD-1 and more messages in both the System D and centralized schemes. SDD-1 needs to know the data requirements of each transaction before it executes. The designers took advantage of non-uniform data access patterns by defining transaction classes. However, the classes place an additional burden on the processing sites as they must either know all the class definitions (takes space) or forward a transaction to another site which can process it (takes time). System D requires many synchronization messages to ensure that transactions are serialized. Note that SDD-1 also requires some synchronization messages (NULLWRITE messages for inactive transaction classes). Messages can significantly increase the time needed to process a transaction because of the communications delay. Too many messages may degrade system performance if the communications medium becomes a bottleneck.

What we desire, then, is the best of both worlds, i.e. efficient conflict resolution at little or no expense to the system. We only want to abort transactions when necessary (non-serializable request sequence) and we want as few synchronization messages as possible. And we do not want to make the system so complex that the structures needed to implement it become a burden on the processing sites. Our proposed solution addresses these problems.

4. Passive Concurrency Control

4.1. Overview

In this section, we describe our "passive" concurrency control algorithm for distributed databases implemented on a local broadcast network. As shown later, no explicit synchronization messages are needed. To review, we assume that transactions are small and access only a few entities during their lifetime (an entity being a tuple or record). We also assume that data is not duplicated. In our scheme, the network is composed of three types of nodes: data nodes, transaction nodes and concurrency control nodes (see Figure 1) Transaction nodes provide a user interface and control execution of transactions. The general

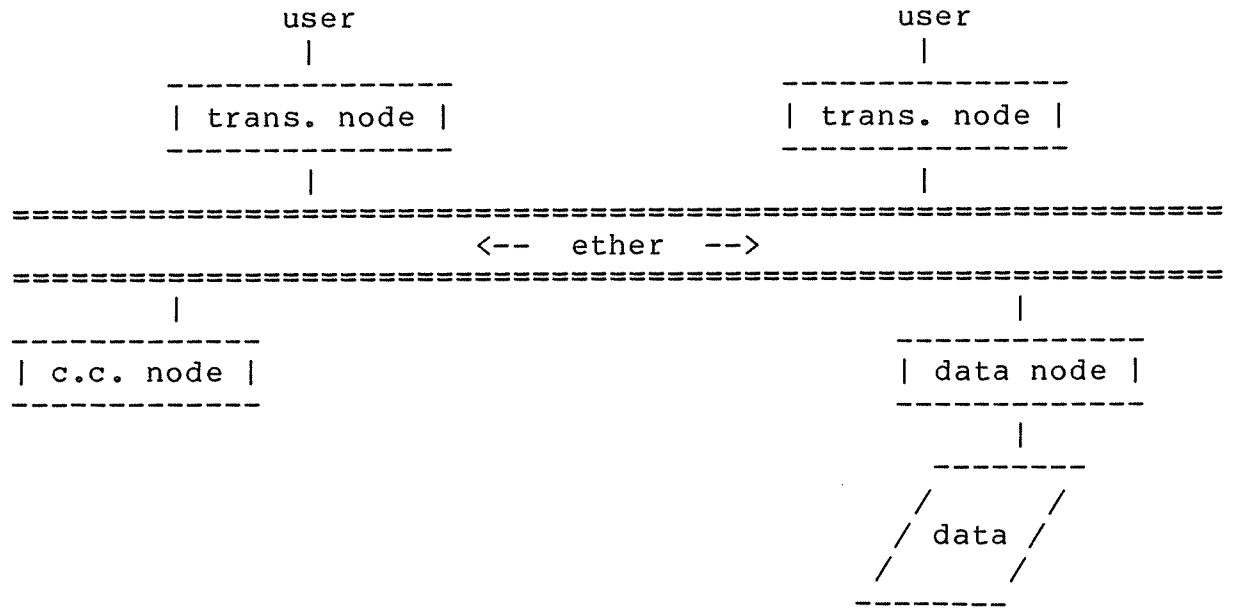

Figure 1. System Configuration

idea is that transactions execute concurrently at transaction nodes and request data and send data as needed from and to data nodes. Data nodes control access to a portion of the database and maintain the necessary indices to insure efficient access to individual records. They maintain a local workspace for each transaction which keeps their working data private until after commit. Entities are not locked and data nodes have no power to reject a request. A read request returns the current value of an entity. A write action updates the value of an entity in the transaction's workspace. A commit action makes an entity's workspace value the new current value for that entity. A concurrency control node monitors the sequence of requests on the ether, maintains conflict information for the transactions, and periodically checks for non-serializable requests sequences (see Figure 2). When found, one of the participating parties is aborted. Otherwise, concurrency control takes place during the transaction commit processing.

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

Requests	A	B
T1: write A	10	--
T2: write B		20
T1: write B		10
T2: write A	20	
	20	10

Figure 2. Non-serializable Request Sequence

The lifecycle of a transaction is, then, as follows. It begins execution at a transaction node which broadcasts a "start transaction" message. Data and concurrency nodes receive the "start trans" message and set up workspaces for the new transaction. The transaction submits read and write requests to various data nodes during execution. It keeps track of which data nodes are participating in the execution by noting who responds to its read and write requests. Each data node scans all read and write requests but processes only those which access part of the database it controls. Since data is not replicated, only one data node will ever respond to a request. When the transaction is ready to commit, the transaction node broadcasts an "OK-commit?" request. The concurrency control node overhears the commit request and takes control of transaction execution at this point. It implements the two-phase commit procedure [Gray78]. The concurrency control node must receive an acknowledgement (of the "OK-commit?") from each data node participating in the transaction before it may broadcast a "commit" message. When a data node gets an "OK-commit?" request, it moves the workspace for the transaction to stable storage [Lampson78], (which makes the transaction recoverable) and acknowledges it. If a data node has crashed, the transaction must abort or wait for the node to recover.

After the concurrency control node has received a commit acknowledgement from each data node participating in the transaction, it must determine if the transaction may safely commit according to the serialization constraints. For example, assume

transaction T2 writes some entity which transaction T1 has concurrently read. The serialization order for these transactions must be T1, T2. Otherwise, T1 would read the entity value written by T2. For now, let us assume that readers of an entity have priority over writers. Then, a transaction may not commit until all readers of entities it has written are finished (Figure 3). If no conflicting transactions (i.e. readers) are found, the transaction may immediately commit. Otherwise, additional measures must be taken to ensure that the end result is the appearance of a serial execution of transactions. These measures are discussed below.

The object of concurrency control is to schedule transactions in a way equivalent to some serial execution of the transactions. Most concurrency control schemes accomplish this by controlling access to the data. But, in our scenario, concurrency control is "passive" (there are no locks hence and access to data is uncontrolled) so we cannot do this. Thus, as in System D, we ensure serializability by controlling the commit process. When a transaction tries to commit "out of order" (e.g.

Transactions: T1: read x, write y T2: read w, write x

Requests:

.
.

T1: read x
T2: write x
T2: OK-commit?

Figure 3. Read-Write Conflict

T2 in Figure 3) we must make it appear as though the transactions committed in order. There are several ways of doing this. A straightforward approach is to cause a transaction to wait until those transactions which must serialize before it have completed. Thus, transactions actually commit in their serialization order. The problem with this approach is that a transaction may have to wait forever to commit. For example, consider a transaction which wants to write entity x. It must wait for all transactions which have read x to finish (since a writer of an entity serializes after readers of an entity). But, there may be an infinite supply of transactions which read x (because data nodes have no power to reject a read/write request). Thus, we have a classic readers-writers problem and our algorithm corresponds to the readers-writers solution which gives priority to readers and lets writers starve. We will refer to this solution as the starvation solution.

The starvation solution suggests another solution: the non-starvation solution. This corresponds to the readers-writers solution in which writers have priority, i.e. an incoming writer only waits for existing readers to finish. Readers arriving after the writer must wait. On our architecture, this algorithm would be implemented similarly to the starvation solution. But, when a transaction wants to commit it is tagged as "golden" and the serialization order is fixed. Then, if subsequent requests cause conflicts which result in a new transaction being serialized before the golden one, that transaction must be aborted. There are several problems with this approach. First is the

additional bookkeeping required by the concurrency node. But since we assume that concurrency control is handled by a dedicated node the cost can perhaps be ignored. A second problem is that this scheme unnecessarily aborts transactions. For example, suppose a transaction writes entity x and tries to commit but must wait for a few readers to finish. Then along comes a short transaction which only reads x and tries to commit. The short transaction would be aborted because it must serialize before the golden transaction. But a transaction which does not produce any output (i.e. does not modify the database) should be able to commit at any time and not affect the serializability of the schedule. Unfortunately, since transactions do not pre-claim their resources we do not know if a transaction will subsequently touch something read or written by a golden transaction. To be safe, we must abort it. A similar problem occurs with the Wound-Wait-Die algorithms.

A third approach to controlling the commit process might be called the restrictions list solution. Recall the original problem. A transaction wants to commit but there are other active transactions which serialize before it. The difficulty is that we can not allow the transaction to commit immediately because subsequent requests from active transactions may result in a cycle. For example, suppose a transaction reads entity x and writes entity y and tries to commit. And, suppose there is another active transaction which has read y. Then that transaction must serialize before the one to commit. But what happens if we let the first transaction immediately commit? We can get

into trouble if the active transaction subsequently tries to write x. This would cause a cycle in the conflict graph which is non-serializable. So, if we allow transactions to immediately commit, we need a way of detecting when later requests cause cycles in the conflict graph. The solution is to include "restriction" lists in the workspace of each transaction. This list restricts what the transaction may access. In the above example, we would allow the transaction to commit but restrict the active transaction from reading x. If the transaction later tried to read x it would be aborted. This idea was proposed by the designers of System D and this would be a non-distributed implementation of it. It is more complex than the previous solutions and would require additional workspace for each transaction. However, since we assumed that transactions are short, the restriction lists should not become unmanageable. This solution is optimal in the sense that committing transactions never have to wait during the commit process for other transactions as in the starvation solution. Also, transactions are never unnecessarily aborted as in the non-starvation solution. Transactions are only aborted when they cause cycles in the serialization graph.

To review, suppose we have the following sequence of requests from three transactions, A, B and C:

```
A: read x
B: read y
A: write y
A: OK-commit?
C: read y
```

The differences between the three algorithms can be summarized in

the way they handle the commit request from A.

starvation: transaction A may not commit until all readers of y have finished: it must wait for B and C to terminate.

non-starvation: transaction A only waits for readers which exist before it tries to commit: it waits for B but C is aborted when it tries to read y.

restrictions list: transaction A may immediately commit, but B and C are prohibited from writing y or writing x.

In all the solutions mentioned above, there is a glitch period between the time the concurrency node decides to commit a transaction and the time it finally broadcasts the commit message. Consider the starvation solution. Suppose a transaction requests a commit and the concurrency controller finds no conflicts. But before it can broadcast a commit message, another read message is overheard which conflicts with the committing transaction. We can no longer commit the transaction since the new reader has priority and should serialize first. We must wait for the new reader to finish. Similar problems occur with the non-starvation and restrictions lists algorithms. The difficulty is that the message input buffer of the concurrency controller is a history of accesses, not a queue of requests. One possible solution is to require the concurrency controller to completely empty its input message buffer before broadcasting a commit message. By doing this, it guarantees that its conflict information is current.

To review, then, a transaction begins life at a transaction node which submits read and write requests on its behalf to data nodes. A concurrency node monitors requests and aborts transac-

tions which have non-serializable conflicts. The data nodes maintain separate workspaces for each transaction until the commit point when the entity values in the workspace become the current values for the entity. When it is ready to finish, a transaction node broadcasts a commit request. A concurrency node hears the commit request and waits for affirmative responses from all the data nodes of the transaction. Then the concurrency node decides to commit the transaction or cause it to wait depending on which of the commit procedures described above (starvation, non-starvation, restrictions list) is used. The transaction node waits for a commit or abort from the concurrency node and terminates the transaction.

4.2. Feasibility

In order to determine if this approach to concurrency control is feasible (with respect to performance requirements), the following simple analysis was made. One important question is whether or not the concurrency control node can keep pace with the amount of message traffic on the ether. If not, no transaction could ever commit. Suppose we require a transaction throughput rate of 100 transactions per second. Also assume that a typical transaction requires n messages to execute. This includes all read, write and acknowledgement messages as well as the commit processing. The concurrency control node would then have to process 100*n messages per second. The message interarrival time is then .01/n seconds. For n=15, this is an interarrival time of 666 microsec., or about 666 instructions on a 1

MIP processor. We need to know the processing requirements for the messages of a transaction. Consider a "typical" transaction which does two reads, two writes and visits two data sites. Such a transaction requires the following messages:

# msgs	description
1	initiate transaction
4	2 read requests and acknowledgements
4	2 write requests and acknowledgements
1	commit request
2	commit acknowledgements from data sites
1	commit
13	total number of messages required

The start transaction message, the read and write acknowledgements and the commit request and commit acknowledgement messages are processed very quickly. This accounts for 8 messages. The read and write requests themselves take longer to process because the serialization graph must be updated which is a closure operation. The more conflicts, the longer this takes, since, for each conflict, you must perform a separate closure operation to find all the serialization constraints it implies. The commit message is sent by the concurrency controller itself but, before sending it, the serialization graph must first be checked. This requires some processing, especially if the graph must be checked several times before the transaction may commit. Also, the conflict matrix and the serialization graph must be updated when the transaction terminates. Thus, 8/13 or about 60 percent of the messages have very modest computing requirements. It is possible they could be handled within 666 instructions. It is not clear

that the remaining messages could be processed that quickly but the numbers are not outside the realm of possibility. Also note that increasing the transaction size (more read and write requests) should not substantially change this since each such request requires an acknowledgement which is processed quickly. So, changing the size of transactions does not substantially change the pattern of message traffic on the ether. Over 50 percent of the messages will be processed quickly. However, more requests from a transaction does increase the chances of a conflict which increases the processing time. In the case of a transaction abort, the serialization graph must be completely rebuilt, which requires much work. It is hoped that aborts will be infrequent enough to not have a detrimental effect.

Another question to be addressed is whether or not the message traffic required for the transactions will exceed the capacity of the ether. Again, we assume a transaction rate of 100 transactions per second with an average of 15 messages per transaction. The majority of messages will be short since they convey little information (read requests, write and commit acknowledgements, commit requests and aborts). The read acknowledgement and write request messages should also be short since we assumed that transactions access record or tuple size entities. Thus, it seems reasonable to assume an average message size of 256 bits (32 bytes), including network header and checksum overhead. An ether with a capacity of 3 Megabits/sec. can handle a maximum of (3 Megabits/sec) / (256 bits/message) = 11.7 messages per msec. So an Ethernet operating optimally could

transmit 11.7 messages per millisecond. We require 1500 messages per second (100 trans * 15 msgs/trans) or a rate of 1.5 messages per msec. This amounts to a utilization of 1.5 / 11.7 or 12.8 percent. On an actual Ethernet, measurement of channel utilization as a function of artificial offered load (percentage of channel capacity) found that channel utilization matched offered load up to 97% of channel capacity [Shoch80]. In other words, contention of the ether did not significantly affect throughput until messages were being sent at a rate equal to 97% of the channel capacity. Thus, an average utilization of 13% seems well within the capability of existing systems and would not result in an extraordinarily heavy load with many collisions. Even if we assume an average message length of 1024 bits, we get a channel utilization of 51% which is acceptable.

5. Research Plans

The first step of our future research is to find concurrency control and recovery algorithms which perform well on local broadcast networks. A subsidiary problem is then to develop evaluation methods which can be used to compare the existing algorithms. Measures of interest include transaction throughput, number of synchronization messages, number of aborts for concurrency control reasons. We have proposed three algorithms: starvation, non-starvation and restrictions list. We propose to compare the algorithms directly by implementing them on an emulated Ethernet which has been implemented in Modula. We also hope to evaluate other concurrency control algorithms proposed

for local broadcast networks (e.g. System D) by emulation on the emulated ethernet in order to compare the performance of our scheme.

When the concurrency control algorithms have been investigated, the next step will be to make the system more robust by incorporating a recovery scheme and specifying protocols to handle lost and garbled messages. For example, an alternative to having the data nodes maintain separate workspaces for each transaction and performing backout on their own is to keep their structure relatively simple. Their only atomic actions would be read and write an entity and they would have no concept of transaction. A read action returns the current value of an entity while a write action creates a new current value of an entity. Thus, the data nodes become little more than intelligent disk controllers. Logging and recovery would be handled by a dedicated node which monitors requests on the ether like the concurrency control node. When a transaction aborts, the node would be responsible for backing out the transaction by sending write messages to restore any entity values written by the transaction. This scheme complicates the work of the concurrency controller. For example, repeatable reads are no longer guaranteed by the data nodes. Also, we cannot simply undo a transaction by blindly undoing all its writes. Suppose transaction T4 writes entity y, then T5 comes along and writes y. Then T4 aborts. We cannot just restore y to its "pre-T4" value because T5 has already written it and may successfully commit. It seems that some communication between the concurrency control node and the recovery node

is needed to correctly back out a transaction.

The problem of lost messages cannot be ignored, especially since the Ethernet was not designed to be a reliable network. Higher level protocols are needed to guarantee that messages are delivered. For example, if a write request is missed by the concurrency control node but heard by the data node, important conflict information is lost. One possibility is to have the concurrency controller acknowledge all read and write requests. But this increases the message traffic. Another possibility is to have the transaction include a message count in its commit request. If the concurrency controller did not have the same count for the transaction, presumably, it missed a message. There may be other protocols worth investigating as well.

An obvious criticism of our approach is that concurrency control is centralized. A failure of the concurrency control node would crash the entire system. However, the approach is no more subject to failure than a single processor running a multi-user database management system or a tightly-coupled multiprocessor database machine with centralized control.

There are several possible solutions which we propose to investigate. First, reliability could be increased by running several concurrency nodes in parallel. Another approach is to have the remaining sites nominate a new concurrency control node in the event of a concurrency control node failure as is done in [Menasce8Øa]. The conflict information could be reconstructed by examing the workspaces at the data nodes. The passive concurrency control scheme might be used in a database machine where

the components are connected by an Ethernet [Boral79a]. Another use of this approach might be a network of Ethernets connected by Gateways. The individual ethers might each have their own passive concurrency controller. Here, a concurrency control node can only crash one Ethernet, not the entire system. Of course, concurrency control would be complicated for transactions which execute across a Gateway. At any rate, more work must be done in order to determine if the passive concurrency control scheme is a practical idea.

6. References

[Boral79a] Boral, H., and D.J. DeWitt, "Design Considerations for Data-flow Database Machines," Proceedings of the ACM-SIGMOD 1980 International Conference of Management of Data, May 1980.

[Boral79b] Boral, H., and D.J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," To appear: ACM Transactions on Database Systems. Also Computer Sciences Technical Report No. 368, University of Wisconsin, October 1979.

[DeWitt79] D. J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Data Base Management Systems," IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979.

[Eswaran76] Eswaran, K.P., Gray, J.N., Lorie, R.A., and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," CACM, Vol. 19, No. 11, November 1976.

[Eswaran80] Eswaran, K.P., Personal communication, April 1980.

[Gray78] Gray, J.N., "Notes on Data Base Operating Systems," IBM Research Report, RJ2188 (30001), February 1978.

[Lampson78] Lampson, B. and H. Sturgis, "Crash Recovery in a Distributed System," Xerox Palo Alto Research Center Technical Report, 1978.

[Menasce80a] Menasce, D., Popek, G., and R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases," ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980.

[Menasce80b] Menasce, D.A., and T. Nakanishi, "Optimistic vs. Pessimistic Concurrency Control in Database Management Systems," To appear in the Proceedings of 1980 Internationl Conference on Very Large Databases, 1980.

[Metcalfe76] Metcalfe, R.M., and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM, Vol. 19, No. 7, July 1976.

[Rosenkrantz78] Rosenkrantz, D.J., Stearns, R.E., and P.M. Lewis, "System Level Concurrency Control for Distributed Database Systems," ACM Transactions on Database Systems, Vol. 3, No. 2, June 1978.

[Rothnie79] Rothnie, J.B., Bernstein, P.A., Fox, S.A., Goodman, N., Hammer, M.M., Landers, T.A., Shipman, D.W., Reeve, C.L., and E. Wong, "SDD-1: A System for Distributed Databases," Computer Corporation of America Technical Report CCA-02-79, January 1979.

[Shoch80] Shoch, J.F., and J.A. Hupp, "Performance of an Ethernet Local Network -- A Preliminary Report," Proceedings of IEEE COMPCON Conference, 1980.

[Stonebraker76] Stonebraker, M., and E. Neuhold, "A Distributed Database Version of Ingres," Univ. Calif. - Berkeley Memorandum No. ERL-M612, September 1976.

[Thomas79] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979.