SLAC-PUB-2360
July 1979
(N)

CENTRALIZED CONTROL UPDATE ALGORITHMS FOR DISTRIBUTED DATABASES

Hector Garcia-Molina

Computer Science Department

Stanford University

and

Stanford Linear Accelerator Center

Stanford, California 94305

ABSTRACT

The problem of updating replicated data in a distributed database will be discussed. Several centralized control algorithms that solve the problem will be presented. They range from a totally centralized algorithm to one which only centralizes the control of the data. The performance of these algorithms is compared for completely duplicated databases in a no failure, update only environment. The algorithms are studied through simulations as well as by an analytic technique based on a queueing model.

1. INTRODUCTION.

In a distributed database, data may be replicated at several nodes of the system. One of the reasons for replicating data is to improve its

(Presented at the 1st International Conference on Distributed Computing Systems, October 1-5, 1979, Huntsville, Alabama).

* Work supported in part by the Department of Energy under contract number DE-AC03-76SF00515, Advanced Research Projects Agency of the Department of Defense under contract MDA903-77-C-0322, and Biotechnology Research Program of the National Institute of Health under Grant NIH PR-00785.

availability. Another reason is to distribute the load by allowing transactions to read the data at different sites. The price that must be paid for the increased availability and the option of concurrent reads at different nodes is an increased cost for processing updates. Updating replicated copies of data is more expensive than updating a single copy of the data because in the replicated case updates must be performed on all copies. Furthermore, it is harder to coordinate conflicting updates when there are multiple copies to be modified than it is to coordinate the updates when there is a single copy to be updated.

In this paper, we will not study the tradeoffs involved in replicating data. We will assume that the decision to replicate a subset of the data has been made. That is, it is either imperative that the data be available even in the face of failures, or it is expected that the number of updates to the data will be considerably smaller than the number of reads on the data. Once we decide to replicate the particular subset of the data, we need to design an algorithm for performing the updates. This algorithm must make sure that all updates are performed on all copies of the data in the system. The algorithm must also guarantee the consistency of the data [2]. Many such algorithms have been suggested, and in this paper we would like to present some of these algorithms and compare their performance. We will concentrate on a particular type of algorithm, the centralized control algorithms. These algorithms are fairly simple and, surprisingly, perform rather well, as we will see shortly.

2. THE MODEL.

In order to study the update algorithms and their performance, we choose a very simple model for the distributed database and the updates. Since we are interested in updates to a particular subset of replicated data, we will view our "system" only as the replicated data and the nodes where it is located. That is, in our "system", all nodes will have a complete copy of the database. We will assume that all transactions that are processed in the system are update transactions.

We will view the database simply as a collection of named items. Each item "i" has associated with it a set of values; each of these values is stored at a different node in the system. We represent the value of item "i" at node x by d[i,x]. The values for a given item should be the same (i.e., d[i,x] should equal d[i,y] for all nodes x, y). However, due to the updating activity, the values may be temporarily different.

In our system, an update transaction A consists of three steps:

(1) Update transaction A requests values for items i1, i2, ..., in. These values are read at any node in the system. That is, we read d[i1,x], d[i2,x], ..., d[in,x] at some node x.

(2) Using the values obtained, A performs some computations and comes up with a set of new values for a subset of the items read i1, i2, ..., im, where m is less than or equal to n.

(3) The new values produced are stored at all nodes in the system. That is, we do " d[ik,x] := new value for item ik " for all nodes x and all items ik in i1, i2, ..., im.

Notice that updates initially specify their read set. Except for this restriction, our update model is a general one. At the end of this paper we will briefly comment on the effect of this restriction.

Finally, in this paper we will assume that no failures occur in the system. This is a strong statement, but we make it in order to simplify the presentation and the analysis of the algorithms. However, the results we obtain here can be extended to the case where failures occur. Due to space limitations, we will be unable to give the details for this here. We will only make a few comments at several points in the paper as to how failures can affect the performance of the algorithms, and we will refer the reader to [5] for a complete presentation.

3. THE COMPLETE CENTRALIZATION ALGORITHM (CCA).

The first update algorithm that we will present is a complete centralization algorithm, CCA (also called a primary copy algorithm [1]). The basic idea of this solution is to select a "central" node where all update transactions are totally executed. The central node then broadcasts the new update values produced by the transactions to all other nodes. A sequence number is attached to each "perform update" message (i.e., the message with the new values) so that the values are stored at each site in the same order that they were produced by the central node. This algorithm provides consistency because all update transactions are serialized by the central node.

We now give a brief outline of the CCA algorithm:

(1) Update transaction A arrives at node x from a user.

(2) Node x forwards transaction A to the central node.

(3) When the central node receives an update transaction A, it places it in a queue. Transactions from this queue are executed one at a time at the central node. That is, the values requested by A are read from the local database, the computations are carried out, and the new values are stored in the local database. (Update transactions can be executed in parallel at the central node as long as a local concurrency control guarantees that the effect on the database is as if transactions were performed one at a time.) A sequence number is assigned to transaction A. This number represents the order, with respect to other transactions, in which A was executed.

(4) "Perform update" messages are sent out by the central node to all other nodes giving them the new values that must be stored at each site. The sequence number of A is appended to these messages.

(5) When a node y receives a "perform update" message, it waits until it has processed all "perform update" messages from transactions with lower sequence numbers. Then node y stores the new values into its local database, as indicated by the message.

There are two potential disadvantages with this algorithm. The first problem is that if the central node crashes, then no more update transactions can be processed. However, this is not really a problem because the complete centralization algorithm (as well as the other algorithms we will present) can be made resilient. The main idea is to have a protocol for electing a new central site when the old central node crashes. The new central node can collect all the state from the active nodes, and based on this, it can complete any unfinished update transactions and start processing new ones. The techniques for making the CCA algorithm crash resistant are given in [5]. When we study the performance of the CCA algorithm, we will use the simple algorithm given above, but as we have stated, the results can be extended to the resilient version.

Another potential problem with the CCA algorithm is that all update transactions must be processed at a single node. This creates a bottleneck which can significantly degrade performance. This paper will show when the bottleneck occurs and how serious a problem it is.

4. THE PERFORMANCE MODEL.

In order to study the performance of the CCA and other algorithms, we use a simple performance model which represents the principal characteristics of a distributed database system. The performance model is described in [7]. Here we will only give a brief outline of the model and its parameters.

Our performance evaluation of the update algorithms does not only count the number of messages transmitted in order to process an update transaction. Our model also takes into account the IO and CPU processing time required by the transactions, as well as the queueing delays involved in waiting for the IO and CPU resources. In addition to this, the performance evaluation also considers the extra delays and processing loads caused by update transactions that conflict.

The main parameters of the performance model are:

(1) The mean interarrival time of update transactions at each node, Ar. The arrival of transactions at each node is a Poisson process.

(2) The average number of items read by an update transaction, Bs. The number of items referenced by a transaction is exponentially distributed with mean Bs. All items are equally likely to be referenced by a transaction. Out of the items read, a random fraction will be modified.

(3) The number of items in the database, M.

(4) The number of nodes, N.

(5) The network transmission time, T. We assume that the time it takes any message to go from one node to any other node is a constant T. (However, the correct operation of the algorithms does not depend on this fact.)

(6) The CPU time needed to set or check a lock (or to check a timestamp), Ct. This parameter is only used in the algorithms that use locks or timestamps.

(7) The CPU compute time, Cu. After an update transaction reads the z values it needs, it will use z times Cu seconds of CPU time in order to produce the new values for the update.

(8) The IO time needed to set or check a lock (or to read or write a timestamp), It. Again, this parameter is only used in the algorithms that use locks or timestamps.

(9) The IO time needed to read or write one item value from a database, Iu.

5. THE PERFORMANCE RESULTS FOR THE CCA ALGORITHM.

The performance of the CCA algorithm was studied using the performance model we have described. The results we present were obtained using a new iterative technique based on queueing theory [3]. The results were also verified through detailed simulations.

The main measure we use for performance evaluation is the average response time of update transactions, R. We define the response time of a transaction as the difference between the finish time and the time when the transaction arrived at its originating node. We consider the transaction to be finished when the originating node has finished all work on the transaction.

Curve "CCA" of Figure 1 shows the average response time of update transactions with the CCA algorithm, as a function of the transaction interarrival time Ar, for a set of representative parameter values. Notice that as Ar decreases, the arrival rate of transactions and the load increases. In this curve we observe a sharp knee which occurs when the central node is swamped by requests to process transactions.

In order to provide a point of comparison, in Figure 1 we also show the performance of another well know update algorithm. This is the distributed voting algorithm (due to Thomas [8]). The average response time of update transactions with this algorithm is given by curve "DVA" in Figure 1. This algorithm does not have a central node which acts as a bottleneck, but surprisingly, its performance is not as good as that of the CCA algorithm. The main reasons for this relatively poor performance of the distributed voting algorithm are that (a) transactions must visit a majority of nodes (instead of one) before being executed, and (b) the CPU and IO loads produced by a voting operation at a node are considerable, while in the CCA algorithm there is no IO and very little CPU load caused by the serialization of updates.

Although it is not shown in Figure 1, both algorithms saturate at about the same interarrival time. When the loads become very high, the analysis is not very accurate and the simulations are very expensive to run. Fortunately, we are not very interested in this region because both algorithms perform so poorly there. For all cases which are not close to the saturation point, the CCA algorithm performs better than the distributed voting algorithm.

The results of Figure 1 are for the particular set of parameter values shown in the figure. Extensive tests have been run to study the effect of the parameters on the average response time. We have found that the CCA algorithm performs better in most cases of interest. The actual difference in average response time between the two algorithms can be reduced or increased by varying some parameters, but the basic relationship remains unchanged. For a two or three node system and for a small value of the It parameter (i.e., the IO time to read or write a timestamp), the performance of the two algorithms is very similar. As the number of nodes N, the transmission time T, or It increases, the difference in average response time increases and the CCA algorithm becomes more attractive. Notice that the results of Figure 1 are for an IO bound situation. However, the results are similar for a CPU bound case.

6. A CENTRALIZED LOCKING SOLUTION.

Since the CCA algorithm performs so well, we will now investigate other centralized approaches in order to try to improve the performance further. If we look at the CCA algorithm, we realize that the central node is the first to saturate. If we can somehow reduce the load at the central node, the knee of the average response time curve should occur at a higher arrival rate of updates, and the update algorithm will be able to process more transactions.

In the CCA algorithm, the central node is performing two distinct functions: (a) the central node is reading the data and performing the computations for all update transactions, and (b) the central node provides the necessary concurrency control for the transactions (i.e., it serializes the transactions). In the algorithm we will propose now, the centralized locking algorithm (CLA), we will move function (a) to the other nodes in order to reduce the load at the central node. Function (b), which is naturally performed at the central node, will remain there.

In the CLA algorithm, the central node will provide concurrency control by managing locks for the items in the database. Before an update transaction is executed, it will request locks for the items it references. When the locks are granted, the transaction will be able to proceed knowing that no other update transaction will interfere.

In the CLA algorithm, an update transaction that arrives at node x is processed as follows:

1) Node x requests from the central node locks for all the items referenced by the transaction.

2) The central node checks all of the requested locks. If all can be granted, then a "grant" message is sent back to node x. If some items are already locked, then the request is queued. There is a queue for each item and a request only waits in one queue at a time. To prevent deadlocks, all transactions request locks for their items in the same predefined order.

3) Once node x gets all of the requested locks, it can proceed with the transaction. The items are read from the local database, and the update values are computed. A "perform update" message is sent to all other nodes informing them of the update. Node x updates the values stored in its local database.

4) When the other nodes receive "perform update" messages, they perform the indicated update on their copy of the database. When the central node receives the "perform update" message, it also releases the locks of the involved items. Requests that were waiting on those items are notified and can continue their locking process at the central node.

To prevent timing problems (e.g., "perform update" messages arriving out of order at a node), the central node gives sequence numbers to all transactions it grants locks to. Nodes must remember the sequence number of the latest update message they have processed and they must delay processing "perform update" messages that are out of order.

7. SEQUENCE NUMBERS PRODUCE UNNECESSARY DELAYS.

The centralized locking algorithm as stated above may produce unnecessary delays in update transactions due to the sequence number restriction. An example is the best way to illustrate this problem.

Suppose that a large update transaction (i.e., one involving many items) arrives at node 1. A lock request is sent to the central node. At the central node, the locks are granted and the transaction is assigned a sequence number, say number 10. The grant message is sent to node 1 where the transaction is executed (assuming that node 1 has processed all updates with sequence numbers less than 10). Executing transaction 10 consists of reading all items in its read set and doing some computations with the values read. Since we assumed that this transaction referenced many items, executing the transaction at node 1 will take a long time.

Suppose that while transaction 10 is being executed at node 1, another transaction arrives at node 2. Node 2 sends a lock request to the central node. Let us assume that this new transaction has no items in common with transaction 10 or any other transactions which are still in progress. Then the central node can grant the requested locks and assigns sequence number 11 to this transaction. A grant message is then sent to node 2 indicating that it can proceed with transaction 11. But node 2 will not be able to execute the transaction because it has not seen transaction 10 yet (i.e., because of the sequence number rule). However, we know that transactions 10 and 11 have no items in common and that they could be performed concurrently. Unfortunately, node 2 does not know this fact.

As far as node 2 knows, the following sequence might have occurred: The locks of transaction 10 were granted, the update performed at all nodes except node 2 and the locks released at the central node. The "perform update" message to node 2 (step 4 in the CLA algorithm) has been delayed and is on its way. Then transaction 11 arrived. It conflicts with transaction 10, but since the locks of transaction 10 have been released, transaction 11 can proceed. Thus transaction 11 has obtained its locks but it cannot be performed at node 2 until node 2 has performed update 10.

Going back to our original situation, if we want node 2 to be able to proceed with transaction 11 while transaction 10 is being executed at node 1, we must give node 2 additional information that permits it to distinguish the current case from the hypothetical case where transactions 10 and 11 conflict. This additional information is available at the central node. There are several ways in which the central node can give node 2 this information. In this paper we will discuss two ways in which this can be done. The algorithm that uses the first method (called the WCLA algorithm) will be presented in section 8, while the algorithm that uses the second alternative (called the MCLA algorithm) is given in section 9. (Note: The WCLA algorithm is the "centralized locking algorithm" of [7].)

8. THE CENTRALIZED LOCKING ALGORITHM WITH "WAIT FOR" LISTS (WCLA).

In the WCLA algorithm, the central node keeps track of the last update transaction that referenced each item in the database. In other words, the central node keeps a table, LAST(i), where LAST(i) is the sequence number of the last update transaction that locked item i. Then, when an update transaction A obtains its locks, the central node constructs a "wait for" list for transaction A. This list, which we will call wait-for(A), includes the sequence number of all update transactions that A must wait for before being executed. Wait-for(A) is simply the list of the LAST(i) entries for all items i referenced by A. The wait-for(A) list is appended to the grant message to A's originating node x. Before node x executes transaction A, it must wait until all "perform update" messages for transactions in wait-for(A) have been processed locally. Notice that node x will only wait for transactions whose resulting values are absolutely necessary for executing A. In our example, update transaction 11 will not be delayed by transaction 10 because transaction 10 did not conflict with transaction 11 and hence is not in the wait for list of transaction 11. Wait-for(A) must also be appended to all "perform update" messages for A, so that the new update values produced by A can be stored at all nodes in the proper sequence and without unnecessary delays.

There are two potential overhead sources in the WCLA algorithm. One is the processing that is needed before an update can be performed. That is, before performing an update, a node must check that all "perform update" messages for transactions in the wait for list of the update have been seen.

To do this, nodes need to have a list of the sequence numbers of all previously processed "perform update" messages. This list may be very long, but there are many ways to compact it. Thus, we expect this list to fit in main memory at each node, and the CPU time needed to check the wait for list against this list of performed updates should be relatively small.

A more serious source of overhead is the construction of the wait for lists at the central node. This node must keep a sequence number (i.e., LAST(i)) for each item in the database, and in most cases this information will not fit in main memory. Thus, in order to read or modify this information, the central node must use the IO device. This is undesirable because we are trying to reduce the processing loads at the critical central node.

Figure 1 shows the average response time of the WCLA algorithm for three different values of the It parameter. The It parameter is the IO time needed to set or check a lock, and in the WCLA algorithm this value should include the IO time needed to read and modify the LAST(i) values. Since the LAST(i) information will usually be in the IO device, the value of It will usually be greater than zero. Hence, the lower curve (It = 0) should be considered only as a lower bound for the WCLA algorithm.

As can be seen in Figure 1, it is possible for the WCLA algorithm to perform worse than the simple CCA algorithm. This occurs when the locking overhead becomes larger than the data reading load which has been moved out of the central node. By using caches, the value of It may be reduced, thus making the WCLA algorithm more attractive.

9. THE CENTRALIZED LOCKING ALGORITHM WITH HOLE LISTS (MCLA).

In this section we present an alternative to the WCLA algorithm which does not have the IO overhead at the central node associated with wait for lists. The idea again is to send additional sequencing information with the grant messages, but we choose information which is more easily accessible at the central node.

Let us use the term "hole list" for the list of update transactions in progress (i.e., locks granted but not released) at the central node. (We use the term hole list because each entry in the list is a hole or a missing entry in the list of transactions that have released their locks.) When the locks of an update transaction are granted, the transaction's sequence number is added to the hole list. When an update releases its locks at the central node, its sequence number is removed from the hole list.

Now consider the relationship between an update transaction A which has just obtained all its locks at the central node and the hole list existing at that instant. If update transaction B is in the hole list, then A and B can not have referenced common items (else A could not have gotten its locks). Therefore, A does not have to wait for B. In other words, the hole list existing at the instant when A obtains its locks is a "do not wait for" list because it contains the sequence number of transactions that can be executed in parallel with A. If we append the hole list to the grant message to A's originating node x, then transaction A can be executed at node x even if node x has not performed the updates in the hole list. In our example, sequence number 10 would be in transaction 11's hole list, so transaction 11 will not be delayed.

Notice that there may be other update transactions which are not in the hole list but do not conflict with A either. For example, a transaction C which does not conflict with A, but released its locks before A got its locks is in this category. We then see that the hole list is a partial "do not wait for" list. If we compare the hole list for an update transaction A with a complete list of all the transactions that do not conflict with A, we find that the hole list contains the more recent entries in the complete list. However, the older transactions in the complete list have probably already been processed at all nodes and are therefore not capable of producing delays like the one illustrated in section 7. So the hole list will probably be enough to eliminate almost all unnecessary delays. As a matter of fact, if the transmission delays are uniform (as we assumed in our model), the use of a hole list will eliminate all unnecessary delays. This is true because in this case all the "perform update" messages for transactions not in A's hole list will arrive at A's originating node before the grant message arrives at that node.

In summary, hole lists are used as follows. When an update transaction A obtains its locks at the central node, a sequence number S(A) and a copy of the hole list H(A) are appended to the grant message for A. Transaction A will be executed at A's originating node only when all transactions with lower sequence number than S(A) but not in H(A) have been seen locally. The sequence number S(A) and the hole list H(A) are also appended to all "perform update" messages so that the values produced by A can be stored at all nodes in the proper sequence. That is, before a node y stores the values produced by A, it must have stored all values for updates with lower sequence number than S(A) but not in H(A).

The advantage of the MCLA algorithm over the WCLA algorithm is that the hole list can be kept in main memory and is easy to update. Thus, the IO overhead for locking in the MCLA algorithm is almost zero. (In most cases, the lock table can also be kept in main memory as a hash table.) The disadvantage of the MCLA algorithm is that it does not eliminate all unnecessary delays [4]. But for a system where communication delays have a small variance, the hole list mechanism will eliminate almost all unnecessary delays.

In our performance model, communication delays are constant, so the MCLA algorithm performs very well. The average response time for the MCLA algorithm is given in Figure 1. (The curve is the same as the one for the WCLA algorithm with It = 0.) In a system where communication delays have a large variability, the performance of the MCLA algorithm will surely deteriorate. However, the response time of transactions in all algorithms will be affected, and which algorithm performs better will depend on the type of the communication delays.

10. THE MCLA ALGORITHM WITH LIMITED HOLE LIST COPIES.

In the MCLA (as well as in the WCLA) algorithm we assumed that lists of arbitrary size could be transmitted in messages. In many systems this may not be possible because there is a bound on the number of sequence numbers that can be included in a message. In [4] we have studied in detail a MCLA algorithm with this limitation. We call this algorithm the MCLA-h algorithm, where h is the maximum size of a hole list copy that can be sent in a message. In the MCLA-h algorithm, we still assume that the hole list at the central node can be of arbitrary size. In this section we will briefly mention some of the results obtained in [4].

There are two basic alternatives for dealing with limited hole list copies. One is to truncate the hole list copy for an update transaction A so that it fits in the allotted number of slots in the message. The second alternative is to have the central node delay sending the grant message for A until the hole list copy shrinks in size. Notice that after we copy the hole list into H(A), the copy will shrink in size as transactions release their locks. When H(A) becomes small enough, we can actually send out the grant message together with H(A).

Which strategy performs better depends on how well the central node can predict what transactions will release their locks first. If the central node can predict what transactions will finish first or if h is zero, then it is best to truncate. Otherwise it is best to delay at the central node until the hole list copy shrinks in size. In either case, the maximum difference in average response time of transactions between the two strategies is about T seconds (where T is the time to send one message).

It turns out that a relatively small value of h is sufficient in order to obtain good performance with the MCLA-h algorithm. For example, in Figure 2 we give the average response time of transactions when the delay at the central node strategy is used, for several values of h. (These are simulation results.)

Notice that a value of h of 4 or 5 is enough to make the performance almost equivalent to the performance of the MCLA-infinity algorithm (which we studied in section 9). Of course, at very high loads there will be a difference between the MCLA-5 and the MCLA-infinity algorithms. But we are not very interested in this case because both algorithms are so close to saturation.

11. CONCLUSIONS.

In this paper we have presented two new centralized update algorithms for replicated data (the WCLA and the MCLA algorithms). We studied the performance of these and other algorithms and discovered that the MCLA (or the MCLA-h with small h) algorithm has the smallest average response time in many cases of interest.

The performance results presented in this paper were obtained for algorithms that were not crash resistant. However, it is possible to make all the algorithms resilient [5], and the cost in terms of performance for doing this is roughly the same for all algorithms (including the distributed voting algorithm). That is, the average response time of transactions in the resilient algorithms during no failure periods will be increased by about the same factor for all algorithms (because of a two phase commit protocol which is always necessary to guarantee that updates are not lost [5]). Therefore, the comparisons we have made here are still valid for the resilient algorithms. (We do not consider the performance of the update algorithms during actual failures because we expect these failures to be rare, and we expect that the performance during the failure periods will not affect the average response time of transactions significantly.)

We also assumed that update transactions specified initially the items they referenced, so that it was possible for a transaction to request locks as a first step. In [6] we study several modifications to the MCLA algorithm which allow us to process transactions that do not initially specify the items they need. These modified centralized algorithms still seem attractive as compared to the other distributed algorithms.

12. ACKNOWLEDGMENTS.

Several useful suggestions and ideas were provided by Clarence Ellis, Ramez El-Masri, Jonathan King, Bruce Lindsay, Toshimi Minoura, Daniel Ries, Gio Wiederhold, and others.

This work was partially supported by the Advanced Research Projects Agency of the Department of Defense under contract MDA903-77-C-0322, by the SLAC Computation Research Group of the Stanford Linear Accelerator Center under Department of Energy contract DE-AC03-76SF00515, and by the Biotechnology Research Program of the National Institute of Health under grant NIH RR-00785.

13. REFERENCES.

[1] P. Alsberg and J. Day, "A Principle for Resilient Sharing of Distributed Resources", 2nd International Conference on Software Engineering, San Francisco, California, 1976.

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

[3] H. Garcia-Molina, "Performance Comparison of Update Algorithms for Distributed Databases, Parts 1-5", Technical Note 143, Digital Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, June 1978.

[4] H. Garcia-Molina, "Performance Comparison of Update Algorithms for Distributed Databases, Part II", Technical Note 146, Digital Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, December 1978.

[5] H. Garcia-Molina, "Crash Recovery in the Centralized Locking Algorithm", November 1978, to appear as a Technical Note.

[6] H. Garcia-Molina, "Partitioned Data, Multiple Controllers and Transactions with an Initially Unspecified Base Set", February 1979, to appear as a Technical Note.

[7] H. Garcia-Molina, "Performance Comparison of Two Update Algorithms for Distributed Databases", Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, August 1978.

[8] R. Thomas, "A Solution to the Update Problem for Multiple Copy Databases Which Uses Distributed Control", Report 3340, Bolt Beranek and Newman Inc., July 1976.

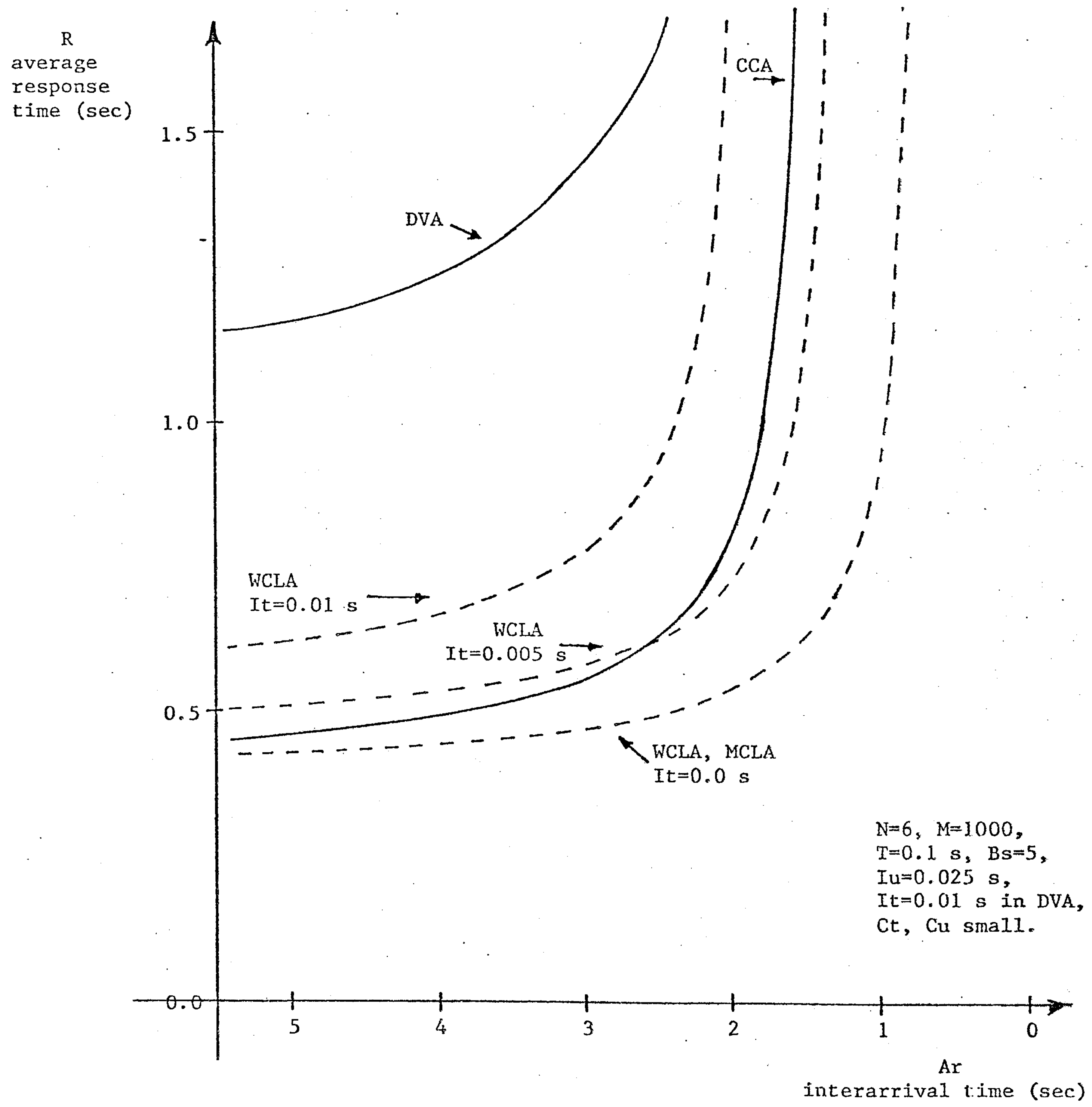
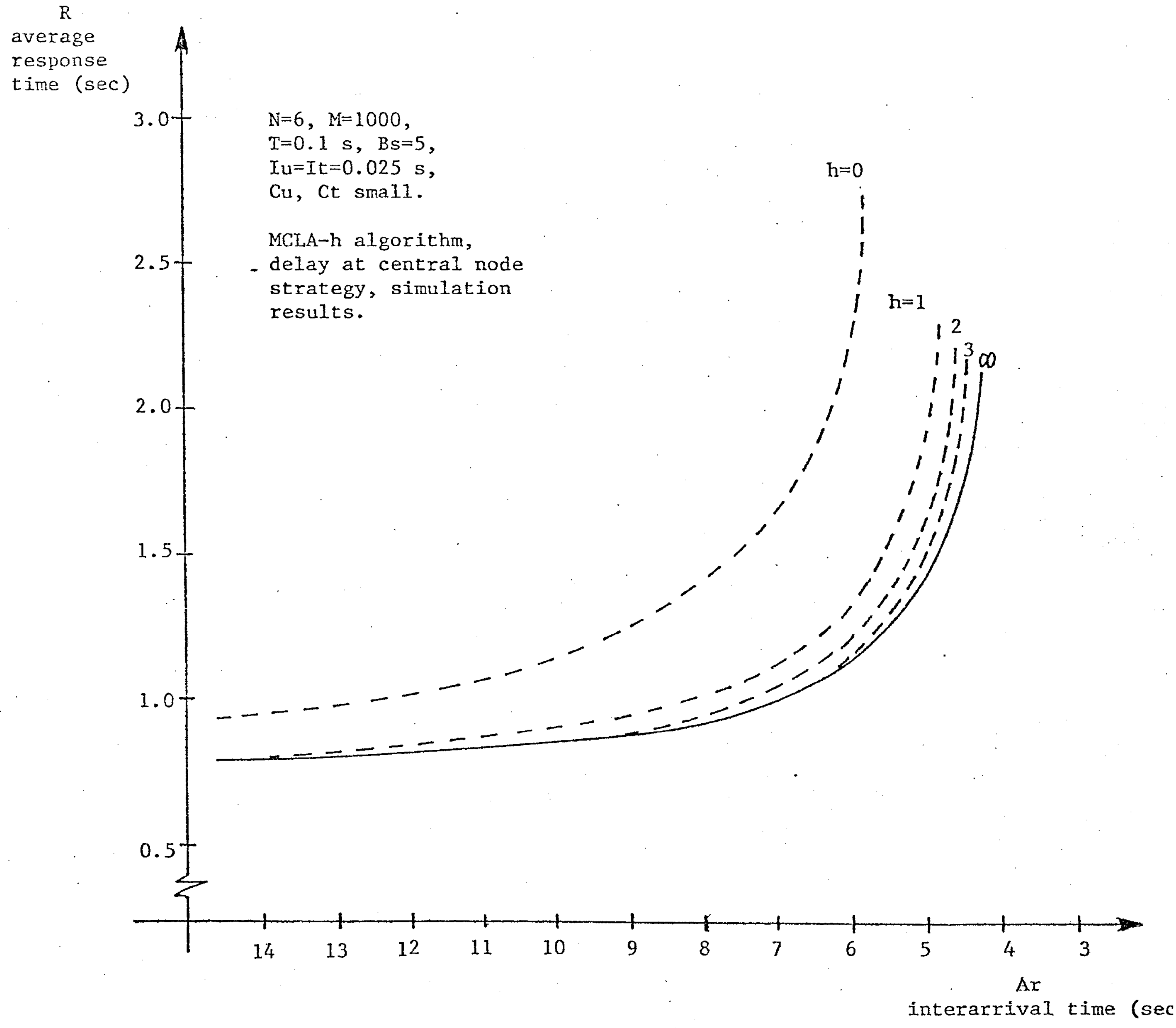

R
average
response
time (sec)
1.5
1.0
0.5
0.0
DVA
CCA
WCLA
It=0.01 s
WCLA
It=0.005 s
WCLA, MCLA
It=0.0 s
N=6, M=1000,
T=0.1 s, Bs=5,
Iu=0.025 s,
It=0.01 s in DVA,
Ct, Cu small.
5
4
3
2
1
0
Ar
interarrival time (sec)

FIGURE 1



R
average
response
time (sec)
3.0
2.5
2.0
1.5
1.0
0.5
N=6, M=1000,
T=0.1 s, Bs=5,
Iu=It=0.025 s,
Cu, Ct small.
MCLA-h algorithm,
delay at central node
strategy, simulation
results.
h=0
h=1
2
3
∞
14
13
12
11
10
9
8
7
6
5
4
3
Ar
interarrival time (sec

FIGURE 2