CONCURRENCY CONTROL AND CONSISTENCY OF MULTIPLE COPIES OF DATA IN DISTRUBUTED INGRES

by

Michael Stonebraker

Memorandum No. UCB/ERL M78/24

24 May 1978

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

CONCURRENCY CONTROL AND CONSISTENCY OF MULTIPLE COPIES OF DATA IN DISTRIBUTED INGRES

by

Michael Stonebraker

Electronics Research Laboratory
University of California, Berkeley

ABSTRACT

This paper contains algorithms for ensuring the consistency of a distributed relational data base subject to multiple concurrent updates. Also included are mechanisms to correctly update multiple copies of objects and to continue operation when less than all machines in the network are operational. Together with [EPST78] and [RIES78], this paper constitutes the significant portions of the design for a distributed data base version of INGRES.

I INTRODUCTION

This paper contains a collection of algorithms to handle concurrency control, crash recovery and the update of multiple copies of relations in a distributed data base system. Together with [EPST78] and [RIES78] this suggests the heart of the design for a distributed version of the relational data base system, INGRES. Earlier thoughts on the same subject are presented in [STON77, WONG77].

The algorithms presented herein are based on what might be called the "primary site" model [ALSB76, STON77] for copies. As such, these algorithms borrow notions from the work of [ALSB76]. Hence, each object possesses a known

---

Research sponsored by Naval Electronics Systems Command Grant #N00039-78-C-0013 and National Science Foundation Grant MCS75-03839-A01.

primary site to which all updates in the network for that object are first directed. It should be clearly noted that different objects may have different primary sites.

In our environment an object is a subset of the rows of a relation (a so called fragment [ROTH77]). Consequently, a relation is partitioned into fragments, each with a primary site and some number of redundant copies. The syntax by which fragments, primary sites and copies are specified is given in [RIES78].

Concurrency control is, of course, a problem to contend with. Unlike [MENA78], who proposes a centralized "GOD" called the lock controller to maintain lock tables and detect and resolve deadlock, we propose to handle locking in a distributed fashion. Deadlock detection and resolution can also be distributed, but we propose allocating these tasks to one machine called the SNOOP. It will be seen that extremely low lock overhead can be achieved using our scheme.

Lastly, crash recovery entails two tasks:

1) handling the failure of a single node in the network and
2) dealing with a failure that partitions the network.

It will be seen that both situations can be handled by very simple algorithms.

In the next section we point out some fundamental assumptions we have made and explain the environment in which distributed data management takes place. Then, in Sections III and IV we present the algorithms for concurrency control and consistency of multiple copies respectively. Lastly, in Section V, we examine our proposal in light of those of [BERN77, MENA78, ROSE77, THOM75].

Throughout this paper we only sketch our algorithms and make no attempt at formal proofs. Arguments along the lines of [MENA78, LAMP76] can be applied to our situation without difficulty, but for the sake of brevity, they are omitted.

II THE ASSUMED ENVIRONMENT

Our work is based on the four following basic assumptions. (The reader can think of these as global assumptions

about the "state of the world". For environments where these are not true, our proposals may not be appropriate.)

1) Locality

At least 95 percent of the traffic to be processed by a distributed data base system is LOCAL. By this we mean that all data to successfully process an interaction is resident at the site where the interaction originates. Moreover, this statement holds even if there are no redundant copies of objects.

We make this assumption based on our belief that other architectural choices (namely centralization or tightly coupled networks with a notion of "GOD") become cost effective when traffic is not predominantly local.

2) Reliability

Computer hardware (and perhaps software) is becoming more reliable. Hence, when possible, mechanisms should make normal processing faster at the expense of lengthened crash recovery time.

3) Closeness

The main performance consideration when processing an interaction is whether it is LOCAL or not. Non local interactions are not expected to be scaled on "closeness", and the notion of a "nearby" site is not assumed to be meaningful. This is a reasonable approximation of most real networks that we are familiar with.

4) Types of networks

The algorithms should work well for both "broadcast" networks and "point-to-point" networks. Hence, they should be able to utilize the capabilities of a network where the cost to send a message to all sites is almost equal to the cost to communicate it to one site. Likewise, the algorithms should also perform well if only point-to-point communication is possible. To a first approximation, we think of the ETHERNET [METC76] and the ARPANET [ROBE70] as generic examples of these two cases.

The algorithms we propose must exist within the context of the query processing done by a distributed INGRES in handling transactions. We now give necessary information on distributed query processing.

A transaction, T, in our environment consists of ONE QUEL command which is either a RETRIEVE or an UPDATE (APPEND, DELETE, REPLACE). There is no extra algorithmic complexity in handling the more general case where a transaction consists of an arbitrary collection of such commands. The only problem is that crash recovery will be decomposed into local crash recovery followed by some extra processing. Current INGRES one-site crash recovery software can only handle the smaller transactions, and we do not plan to rewrite it to be more general.

In the discussion to follow, the reader should note that while we present the restricted context, our algorithms will work for the general case if one pretends that the local notions of "commit transaction" and "recover transaction" handle the general case for a one-site environment.

Such a transaction originates from a user process (application program or interactive terminal monitor) at some site, i, in the network. A "master" or coordinating collection of INGRES processes is invoked at site i. This master creates "slave" INGRES processes at other sites and ensures that each slave knows the identity of all other slaves. Although slaves need be created only at sites where processing will take place, it may be convenient to pretend that they exist at all sites. Hence, we make no further note of creating or destroying slaves. As discussed in [EPST78], a master can send two commands to slaves:

1) run the (local) interaction, I, at a subset of the sites.
2) send a copy of fragment R_j to a subset of the sites in the network.

The slaves eventually return a "done". By a sequence of such commands, as shown above, the master ensures that the user interaction is eventually processed. The details on how such a sequence is generated appear in [EPST78].

If T is a RETRIEVE command then it can be equally thought of as a sequence $I_1,\ldots,I_n$ of interactions where

each I_j is either a local RETRIEVE command or a SEND command to be executed on a subcollection of the sites.

If T is an update, the master first generates a RETRIEVE command to figure out what to do. This command is processed as above and (eventually) generates a deferred update list of the changes to be made. As noted in [EPST78], these lists of updates may not always be at the site where the tuples to be updated reside. Consequently, the lists may have to be appropriately shuffled so that each slave has a complete deferred update list for its site. Since each slave knows the identity of all other slaves, it can simply wait for a message from all other sites containing the appropriate tuples. Alternatively, the master site knows the collection of sites where deferred update is possible. It can send this information to the slaves with some interaction, I. Each slave, in this case, need only expect a message from this subset of sites.

When all is appropriately arranged, a "commit transaction" message must be sent by the master to all slaves. Hence, an update transaction is a sequence of local interactions, $I_1,\ldots,I_n$, followed by some shuffling and a "commit transaction". The detailed protocols to be followed on updates will be explained presently.

It should be noted that $I_1,\ldots,I_n$ for RETRIEVES and updates is always a collection of local RETRIEVES and SENDS. For updates however, the local RETRIEVES that isolate changes to be made sometimes contain a "tuple identifier" as part of the requested data. In this way, the deferred update list will have an indication of exactly which tuples must be modified. The "tuple identifier" also indicates that the current RETRIEVE is really isolating tuples which will later be changed. Hence, for some local locking algorithms (such as [GRAY76]), these must be considered writes rather than reads.

III CONCURRENCY CONTROL ALGORITHM

If T is a RETRIEVE command, proceed according to step one or step two.

STEP ONE: (high performance but has data integrity

problems)

The master directs each interaction, I, to the local copy of each object. If one does not exist, he directs it to the primary copy. Section 4 indicates how the identity of the primary copy is determined.

STEP TWO: (lower performance but no "surprises")

The master directs each I to the primary copy of each object. (We will discuss in Section 4 what the "surprises" are and when step one can be safely chosen.)

Alternatively, if T is an update, the master must direct each I to the primary copy of the object without exception.

Note that the choice of step one or step two will affect whether a particular interaction is local or not. Hence, it may affect the sequence of interactions which is chosen by the master during distributed decomposition.

A local concurrency controller (cc) runs at each site. This cc sees a transaction, T, as consisting of a collection of interactions, $I_1,\ldots,I_n$, EACH LOCAL TO THAT SITE which it receives one by one from the master. It is assumed that level 3 consistency [GRAY76] for such a transaction is satisfied. How cc performs this task is of no concern here. All that matters is that the response to each interaction, I, which a master sends to a slave, is a "done" (indicating that it is complete) or no response (indicating that it is either still processing or is waiting for permission from the cc to proceed further).

Since the sequence of interactions may not be known in advance, deadlock may occur. If it happens within one machine, the cc picks a victim and performs local backout. The appropriate slave sends a "reset" message to his master which in turn resets all other slaves. This accomplishes backing out T.

However, deadlock may happen in such a way that it involves multiple machines. We suggest the following mechanism to deal with this problem. Whenever an interaction is run at a node it will either run to completion or wait, due to a lock conflict. Say transaction Tj holds a lock for which transaction Tk waits. When cc detects such a

condition, it sends identifiers for Tj and Tk to an agreed on machine (The SNOOP). The SNOOP then detects deadlock by ordinary analysis of the global "wait for" graph, which it can generate from all such messages. When a master completes a transaction, it sends a "done" to the SNOOP, who can appropriately update his "wait for" graph.

In summary, each site has a cc that handles concurrency control for local transactions (using some technique). The lock tables each cc creates and uses are LOCAL to its site and are present nowhere else. Only the SNOOP needs to be informed of "wait for" conditions.

In the absence of crashes, it is clear that such a scheme can work correctly. We now turn to control of multiple copies and crash recovery.

IV COPIES AND CRASH RECOVERY

There are two algorithms to handle copies and crash recovery. One is called the "performance" algorithm. It processes updates with the minimum possible delay (i.e. fastest possible response time). However, it will be seen to have a data consistency problem since transactions can be lost and relations can become inconsistent if certain sorts of crashes happen. In the absence of crashes, however, there are no consistency problems.

This algorithm will require the notion of a "copy INGRES". This is an invocation of INGRES that can receive a complete deferred update list from a slave INGRES for some object, and performs the update on a copy of that object. A copy INGRES must exist for every copy of the object which is not the primary copy.

The other algorithm is called the "reliability" algorithm. It ensures that there is never a data consistency problem. However, there may be a substantial response time penalty in order to guarantee this.

In this section we make the following assumptions:

1) Crashes of a single node can happen. Service is restored after some delay. Local crash recovery procedures will automatically be run and will guarantee that local

transactions, which were "committed" before the crash but whose effects were not completely merged into the local data base, are correctly committed.

2) Communication failures can happen. They will either be transparent or will partition the network into subsets that cannot communicate with each other.

3) Every message sent is reliably received if the recipient is "up" and the sender does not crash inopportunely. If the recipient is down, the sender can queue the message for later delivery. Presumably, it is queued at the sender's site so that subsequent delivery can only happen if the sender is "up".

In this section we will indicate several algorithms that must be run by the various actors under various circumstances. The data structures used in the algorithms are the following.

4.1 Data Structures

Each machine in the network maintains the following information:

1) a state flag

This is either "normal", indicating normal operation, or "restructure", which means the network is dynamically adjusting to a crash or restoration of service.

2) an up-list

This is a list of sites that the given node thinks are in operation. These sites form a "logical partition" [MENA78] of the network that may lag in time behind the actual physical state of the network.

3) the identity of the current SNOOP

This is an identifier for the machine that performs deadlock detection. If the up-list changes in such a way that the current SNOOP is not in the up-list, there is a globally agreed upon procedure for choosing a new SNOOP.

Moreover, each master INGRES must have:

4) a commit flag

This flag is set by the master when it thinks that a transaction is committed. It is reset otherwise.

Each slave INGRES must have:

5) a ready flag

This flag is set if the slave has the proper deferred update list and is ready to commit a transaction. It is reset otherwise.

6) a commit flag

This is set when the slave believes that his transaction is committed and is reset otherwise.

Lastly, the location of all copies for each object is known (by an examination of some system catalogs whose composition will not concern us here). In addition, there is a known linear ordering of all these copies. As a result, any site can always access its up-list and find the particular copy which is lowest in the ordering among those at sites in the up-list. This is by definition the PRIMARY COPY of the object.

Now, if the network is currently partitioned, nodes in different partitions will disagree on the identity of the primary copy. Obviously, chaos would result if more than one partition was allowed to independently update a given object, since there would be no possible way to make the objects consistent when the network was repaired. The easiest way to avoid this situation is to allow a primary copy to exist ONLY if a majority of all copies are at sites in the up-list. Otherwise, the object is inaccessible.

However, it should be clearly noted that a problem arises in the case where there are exactly two copies of an object. In this (presumably very common) situation, any network partition will make both copies inaccessible. To handle this, one must be able to tell the difference between:

a) a crash of a single site and

b) a network partition that leaves each copy in a different partition.

In the former case we wish the backup copy to become the primary, while in the latter case we do not. If these two situations cannot be differentiated, the backup copy is worthless.

There are six algorithms to be presented. In order to assist the reader we first outline the general flow of control before proceeding with the details.

ALGORITHM MASTER, SLAVE and COPY are executed respectively by master, slave and copy INGRESes. The key point is that a master has a "commit point" when it sets its commit flag. Before that point, it will back out a transaction if a failure occurs. After that point it will ensure that the transaction is correctly committed. The tactics used amount to a "two phase" commit protocol [GRAY77] and are similar to the proposal of [LAMP76].

Each slave responds to directives from the master. It uses the local commit flag to indicate whether it thinks that the transaction is committed. On crashes, local recovery examines the flag to decide whether to backout or complete the local transaction. When a slave commits a transaction, it in turn directs copy INGRESes to correctly update the auxiliary copies.

The final 3 algorithms are run in the context of failures. ALGORITHM LOCAL RECOVERY performs "local cleanup" and is run when a site wishes to resume service. ALGORITHM RECONFIGURE is used to adjust the up-list after a failure or a service restoration. It allows the network to continue with less than all sites operational or even in a partitioned state. Lastly, ALGORITHM SLAVE PROMOTE is run when a master crashes. Its purpose is to allow the slaves to correctly finish or backout the transaction in question.

4.2 The Performance Algorithms

The reader is again cautioned that data integrity problems exist in the algorithms which are now presented.

ALGORITHM MASTER

This is the algorithm to be used by a master INGRES when it receives a an update transaction, T, from a user process.

m1) refuse T unless state = "normal"

m2) Examine the up-list and calculate the primary site for each object involved in T. Refuse T if the primary site of any object is inaccessible.

m3) Supervise distributed decomposition for T and coordinate getting a correct local deferred update list (du) to each slave.

m4) Wait for a "I am ready" message from each slave. If not all respond, go to m8).

m5) Set commit flag. Send a "commit" message to all slaves. Send a "done" to the user process.

m6) Wait for a "done" from all sites. If not all respond, go to m9).

m7) Send a "done" to the application program (if it is more convenient to wait until master processing is complete). Send a "done" to the SNOOP. Go to m1).

m8) Send "reset" to all slaves, the user process, and the SNOOP. Queue this message for later delivery to the non responding machines. Send a "reconfigure" message to each site. Go to m1).

m9) Queue a "commit" message and a "potential trouble" message for the down machine(s). Send a "reconfigure" message to each site. Go to m1).

ALGORITHM SLAVE

This algorithm is to be run by each slave which assists a master in processing a transaction, T.

s1) Execute interactions at the request of the master and

assemble a deferred update list (du) for the local site. When it is correct, set the ready flag.

s2) Send "I am ready" to the master.

s3) Wait for "commit" or "reset"

s4) If a "commit" is received, then set local commit flag. Commit the transaction. Respond "done" to the master when completed. Spawn a process to execute steps s6) to s8). Go to s1).

s5) If a "reset" is received, run ALGORITHM LOCAL RECOVERY. When complete, go to s1).

s6) Send (du) to each copy INGRES for the object being updated that is operational. Queue (du) for later delivery to non responding sites.

s7) Wait for a "done" from each copy INGRES. If not all respond, queue a (du) message for the non respondants and send a "reconfigure" message to all sites.

s8) Terminate.

ALGORITHM COPY

This algorithm is run by each copy INGRES that is assisting in processing a transaction, T.

c1) Wait for (du).

c2) Perform update and respond "done"

ALGORITHM LOCAL RECOVERY

This algorithm must be run when a site restores service or when a transaction wishes to "reset".

r1) Read all outstanding (i.e. queued) messages and perform appropriate local actions.

r2) If a local commit flag is set (either for the transaction in question or all transactions), commit the update. If not, correctly back out the transaction. Techniques for doing r2) for "soft" crashes, (i.e. ones in which data is not lost from secondary storage), are well known [STON76, ASTR76, LAMP76] and easily implementable. A more elaborate (and tedious) restoration must be done for hard crashes where data may be lost.

r3) If this is a service restoration, then proceed, otherwise, send a "done" to master and terminate.

r4) Send a "reconfigure" message to all sites.

r5) Execute ALGORITHM RECONFIGURE.

ALGORITHM RECONFIGURE

This algorithm must be executed whenever a "reconfigure" message is received by a site. Its purpose is to dynamically alter the up-list.

f1) set state = "reconfigure". Instruct each master INGRES at the local site to backout or complete the current transaction. Go to f2) when the site is quiescent.

f2) Send "I am up" to all sites.

f3) Wait for a predetermined interval for replies and set the up-list equal to the set of all respondants.

f4) If current machine is the lowest in a predetermined ordering of all machines in the up-list, go to f5); otherwise, go to f8).

f5) Send up-list to all sites in the up-list.

f6) Wait for "I agree" from everybody. If any do not respond (or respond with "disagree"), send a "reconfigure" message to each site and go to f1).

f7) Set state = "normal". Send "normal" to all sites on the

up-list and go to f11).

f8) Wait for up-list from some other site.

f9) Compare with local up-list and respond with "agree" or "disagree".

f10) Wait for "normal". When received, set state := "normal".

f11) If current SNOOP is in the up-list, resume normal operation. Otherwise, calculate the new SNOOP by the prearranged algorithm and send local conflict graph to that site. Resume normal operation.

It should be noted that step f1) will take a varied amount of time and an operational site may not send an "I am up" message within the "predetermined interval" of step f3). This problem can be handled by a straight forward (but tedious) extension of the algorithm.

It should also be noted that various multiple crash conditions can leave all up sites at step f8) in the algorithm. This can be detected and handled by a timeout and algorithm restart mechanism.

ALGORITHM SLAVE PROMOTE

This algorithm is executed by any slave that cannot communicate with its master.

p1) invoke execution of ALGORITHM RECONFIGURE

p2) Wait for state = "normal".

p3) Examine up-list to see whether all other slaves are up. (If not, there may be a data consistency problem which is discussed in the sequel).

p4) If the slave is the lowest one in a predetermined ordering of the slaves in the up-list, go to p5); otherwise, go to p9).

p5) Send "is there a commit flag on" to all other up slaves.

p6) Wait for all replies.

p7) If there are no commits, send a "reset" to all slaves, the SNOOP and the user process. Queue this message to down sites.

p8) If there is one or more commits, send a "commit" to all slaves and go to m6) of the ALGORITHM MASTER.

p9) Wait for "is there a commit" message.

p10) Reply "yes" or "no" and go to step s3) of the ALGORITHM SLAVE.

4.3 Data Integrity Problems with the Performance Algorithms

This collection of algorithms has the following three "features".

1) Transactions may be lost

This situation will normally be detected by a received "potential trouble" message. If a slave crashes after it has set its local commit flag but before it has reliably communicated (du) to any copy INGRES, the following situation arises. No copy of the object in question has received the update. However, ALGORITHM LOCAL RECOVERY will ensure that a service restoration at the crashed site will commit the update. Hence, when service is restored and the network is reconfigured, one copy will not agree with the others because it will have had an extra update performed. This "phantom" is lost until service restoration, at which point it generates a consistency problem.

Of course, the slave can make this possibility very remote. However, there is another serious case to consider. Suppose a slave crashes after the master has set its commit flag. In this case, other slaves associated with the same master will proceed with the update, and the crashed site will not. Moreover, no copy of the object at the crashed site will ever be updated. This generates the following feature.

2) The data base may be inconsistent

According to the above scenario, a user process could be giving a ten percent raise to all system programmers. If system programmers appear at fifteen sites, then fifteen slaves will be processing. A crash of one of them at an inopportune time will leave the update committed at fourteen sites and undone at the other. Hence, some system programmers got the raise and some did not.

3) Messages can get lost

The algorithms all assume that a down machine recovers by (among other things) reading its outstanding messages. However, some outstanding messages may be on machines that are currently down. This may mean, for example, that (du) messages that were queued are not delivered before a machine resumes normal operation. Depending on the composition of the lost messages, the result can be benign or catastrophic.

4.4 Reasons for Tolerating Data Inconsistency

Obviously, these are serious drawbacks. Moreover, it is not clear what a user process can do to deal with these problems. However, there appear to be cases where a user might be willing to tolerate such "surprises". Three examples are suggested.

1) Update transactions always affect only one fragment of a relation.

In this case, problem 2 cannot arise and problem 1 can be easily recovered from. If machines are reasonably reliable, problem 3 may not be an issue. (It only arises if at least two machines are simultaneously out of service.)

2) Almost all updates are local.

With the above algorithms, a "done" can be given to the user process for a local transaction after ONLY local processing. Hence, the response time can be (more or less) the same as for a centralized system. To alleviate any of the above problems would require at least one other machine be signalled before a "done" could be sent to the user process. This may degrade response time substantially (by perhaps an

order of magnitude, depending on the speed of the network). There may be situations where such a degraded response time is unacceptable.

3) Almost all updates can be decomposed into a sequence of interactions, each of which is local to some machine. Again, response time may be substantially degraded by any mechanism dealing with the above problems.

As a result, a data base administrator can decide if better response time is worth the above headaches. For administrators who answer "no", we offer a collection of much more reliable (and expensive) algorithms.

4.5 The Reliable Algorithms

The previous algorithms may be made more reliable by making the following changes:

1) Treat each copy INGRES identically as the slave for which it is updating a back-up copy. Hence, each copy INGRES will receive a correct (du) before a transaction is committed. All messages which are sent to slaves are also sent to copy INGRESes.

2) ALGORITHM COPY is no longer necessary.

3) Steps s6) to s8) of ALGORITHM SLAVE are no longer necessary.

4) At every step where the instruction to queue a message for a process at a non responding site occurs, change it to queue the message at K sites, K>1.

With these modifications it is easy to show that all three problems mentioned above are avoided if less than K machines are down at any one time and if there are at least K copies of each object.

The cost of this scheme is, of course, that a "done" cannot be signalled to the user process until a correct (du) is received reliably at ALL sites where a copy resides. Clearly, substantial network traffic (and delay) will be required to perform this.

It is obvious that there is no possible scheme which can guarantee that there will be no "surprises" if K-1 machines crash, unless that scheme ensures that any update is reliably received at K sites before commiting the update. Whether users are willing to pay the cost in response time for such a guarantee is unknown.

4.6 Accesses to the Local Copy

We are now in a position to discuss when a RETRIEVE can be reliably directed to the local copy (step one from Section 3). Clearly a RETRIEVE can be directed to the local copy of an object (step one) if it accesses only one fragment of any relation. If a RETRIEVE accesses multiple fragments of the same relation, then the possibility exists that the local copies of each fragment are not consistent. This will happen if there is an update which spans multiple fragments (such as the previously discussed ten percent update to all system programmers) and some local fragments have received the update and the others have not (since copies are updated asynchronously). This will be termed the "multiple fragment read problem".

Clearly, a multiple fragment RETRIEVE may yield an erroneous answer and there is no way to guard against this possibility (except by synchronizing the updating of copies of fragments). However, since it is a performance winner to access local fragments if possible, we plan to allow the user to set a flag indicating that he wishes to "take a chance" and use the local copy for RETRIEVE commands.

V COMPARISON WITH OTHER SCHEMES

We now examine our scheme in comparison with the updating schemes proposed by [BERN77, MENA78, ROSE77, THOM75].

In [MENA78] a centralized concurrency control scheme is proposed. Moreover, to successfully handle arbitrary crashes, the lock table for this centralized controller must be fully redundant (i.e. present at each of N sites). Therefore, each interaction, I, in our model - using their scheme, requires 3*N-1 messages to correctly alert all N copies of the lock table. Our scheme only requires 1 message (the interaction I) to be sent to a SUBCOLLECTION of

the sites. Moreover, this message must be sent anyway. Only if a "wait for" condition is generated must a second message be sent to the SNOOP. Moreover, equal robustness on crashes is achieved by our scheme with far less messages. In addition, our recovery algorithms are much simpler because the lock table is not redundant (and hence cannot be inconsistent with copies of itself).

[ROSE77] proposes a concurrency control scheme that avoids using the SNOOP by having timestamps on transactions to resolve deadlock in a distributed fashion. However, that scheme may abort a transaction without a deadlock existing. Our scheme, on the other hand, backs out a transaction only if a deadlock actually occurs. Whichever scheme has less overhead is application dependent, and a simulation model (perhaps along the lines of an extension to [RIES77]) would have to resolve the issue.

In [THOM75] a scheme is presented for updating multiple copies of objects based on majority voting. In this proposal, a "done" cannot be given to a user until n/2 copies are serially updated. Only when this number is achieved can the transaction be sure that it will not be backed out due to conflict. Clearly, our performance algorithms generate a "done" much sooner. However, our reliability algorithms (in essence) require all copies to be updated before a commitment is signalled. On the other hand, we deal robustly with arbitrary crashes. [THOM75] does not contain a crash recovery proposal (and would have to be extended to do so). Without adding crash recovery procedures to the algorithms of [THOM75], there is nothing more that we can say.

We now consider the proposal of [BERN77]. In environments where traffic is not predominantly local, it appears to be a desirable (but very complex) solution. Our scheme, on the other hand, is directed toward environments where there is high locality (assumption 1 from Section II).

Without giving the details of [BERN77], it should just be noted that it proposes doing conflict resolution (if possible) at data base design time. If transactions do not conflict, each can update the copies of an object in any order. The preferred order is to update the local copy first and give a "done" to the user process when this is complete. Hence, excellent response time can be guaranteed.

Transactions that cannot be shown to be non conflicting do not have this freedom. For such transactions more extensive coordination must be done. In general this will require communicating with all sites where copies reside, although in fortuitous circumstances a "luck out" may happen and a site will send the right sort of a message without being requested to.

Comparing our scheme with that of [BERN77] we first presume that an equal number of copies of objects exist at the same sites for either situation. Also, we note that the algorithms in [BERN77] are not robust on machine crashes. Providing such a feature will require additional mechanisms. Hence, it appears appropriate to compare [BERN77] with our performance algorithms, which also fail to make robustness guarantees. Lastly, [BERN77] avoids the "multiple fragment read problem" mentioned earlier by requiring that a transaction only send ONE write message to each site (p 13). This will ensure that all fragments of objects at a given site are updated atomically. This requirement either substantially reduces the potential parallelism with which a transaction can be executed or requires extra synchronization effort. In any case, this condition can be equally well guaranteed by our proposal. It should be noted that assuming this notion allows us to direct RETRIEVE transactions to the local copy of every object.

The comparison to follow is based on response time only. The actual overhead of the two schemes depends on what is counted, is stochastic in nature, and would require additional study to quantify. The comparison is based on the following cases:

1) The transaction is a RETRIEVE and there is a local copy of each referenced object.

Here both proposals can access the local object and response time is the same either way (assuming that both proposals synchronize multiple fragment updates as discussed above).

2) The transaction is a RETRIEVE and 1) is not satisfied.

In either case, a non-local copy is accessed. By our assumptions concerning closeness (assumption 3), it does not

matter which copy is used, and the delay is the same.

3) The transaction is an update and the primary copy of all objects is local, and the transaction is one which can be preanalyzed by [BERN77] as a "non conflict" one.

Both algorithms will generate a "done" after local processing only.

4) The transaction is an update and its primary copy of all objects is local and it is not in the "non conflict" category.

Here, [BERN77] requires a message be sent to each site and a response received (unless there is a "luck out"). Our scheme requires only local processing.

5) The transaction is an update and is "non conflicting" but is not local.

Here, [BERN77] requires no messages (but only if there is a local copy of all objects). Our scheme requires communicating with the primary site of each non local object.

6) other cases

This is the same as 4) for [BERN77] and 5) for our proposal.

Suppose X percent of the update transactions are local for our algorithm and Y percent are "non conflicting" for [BERN77]. Figure 1 indicates the number of sites which must be communicated with for each scheme in cases 3)-6). Here, c represents the number of copies of an object which are present in the network.

CASE	PROBABILITY	NUMBER OF SITES TO BE COMMUNICATED WITH PER OBJECT ([BERN77])	NUMBER OF SITES TO BE COMMUNICATED WITH PER OBJECT (OUR PROPOSAL)
3	XY	0	0
4	X(1-Y)	c	0
5	Y(1-X)	0	1
6	(1-X)(1-Y)	c	1

A Comparison of Our Scheme With [BERN77]
Figure 1

CLearly, [BERN77] will win only if the following inequality holds:

$$Y > 1 - \{ (1-X)/c \}$$

By assumption 1 concerning locality, $1-X < 0.05$. Hence, if $c = 3$ then $Y > 0.983$ in order for our proposal to lose. Of course, the actual value for Y is application dependent. To support the plausibility of a high Y, an example is presented in [ROTH77a] where Y=0.9925. However, for the identical example an interested reader can ascertain that $X = 0.9925$ also. A cursory examination of the above equation indicates that our proposal wins for the case Y=X.

REFERENCES

[ALSB76] Alsberg, P. A. and Day, J. D., "A Principle for Resilient Sharing of Distributed Resources," Center for Advanced Computation, University of Illinois, Urbana, Iii, 1976.

[ASTR76] Astrahan, M. M. et. al., "System R: A Relational Approach to Database Management," TODS 2, 2, June

1976.

[BERN77] Bernstein, P. A. et. al., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Data Bases," Computer Corp of America, Cambridge, Mass., December 1977.

[EPST78] Epstein, R. S., et. al., "Distributed Query Processing in a Relational Data Base System," Proc. 1978 ACM-SIGMOD Conference on Management of Data, Austin, Texas, May, 1978.

[GRAY76] Gray J. et. al., "Granularity of Locks and Degrees of Consistency in a Shared Data Base," IBM Research, San Jose, Ca., RJ 1849, July, 1976.

[GRAY77] Gray, J., "Notes on Data Base Operating Systems," unpublished course notes, July 1977.

[LAMP76] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed System," Xerox Palo Alto Research Center, 1976.

[MENA78] Menasce, D. A. et. al., "A Locking Protocol for Resource Coordination in Distributed Systems," Proc. 1978 ACM-SIGMOD Conference on Management of Data, Austin, Texas, May 1978.

[METC76] Metcalfe, R. M. and Bogs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM 19, 7, July 1976.

[ROBE70] Roberts, L. and Wessler, B., "Computer Network Development to Achieve Resource Sharing," Proc. 1970 Spring Joint Computer Conference, 1970.

[RIES77] Ries, D. and Stonebraker, M., "A Study of the Effect of Locking Granularity in a Relational Data Base System," TODS 3, 3, September 1977.

[RIES78] Ries, D. and Epstein, R., "Specification of the Distribution Criteria for a Distributed Data Base

System," Electronics Research Laboratory, University of California, Berkeley, Ca., Memo #M78-25.

[ROSE77] Rosenkrantz, D. J. et. al., "A System Level Concurrency Control for Distributed Database Systems," Proc. 2nd Berkeley Workshop on Distributed Data Bases and Computer Networks, Berkeley, Ca., May 1977.

[ROTH77] Rothnie, J. B. and Goodman, N., "An Overview of the Preliminary Design of SDD-1: A System for Distributed Data Bases," Proc. 2nd Berkeley Workshop on Distributed Data Bases and Computer Networks, Berkeley, Ca., May 1977.

[ROTH77a] Rothnie, J. B. et. al., "The Redundant Update Methodology of SDD-1: A System for Distributed Databases," Computer Corporation of America, Cambridge Mass., June 1977.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON77] Stonebraker, M. and Neuhold, E., "A Distributed Data Base Version of INGRES," Proc. 2nd Berkeley Workshop on Distributed Data Bases and Computer Networks, Berkeley, Ca., May 1976.

[THOM75] Thomas, R. H., "A Solution to the Update Problem for Multiple Copy Data Bases Which Use Distributed Control," BBN Report #3340, July 1975.

[WONG77] Wong, E., "Retrieving Dispersed Data from SDD-1: A System for Distributed Data Bases," proc. 2nd Berkeley Workshop on Distributed Data Bases and Computer Networks, Berkeley, Ca., May 1977.