A ROBUST DISTRIBUTED CONCURRENCY CONTROL SCHEME

Abraham Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-224 April 1983

A ROBUST DISTRIBUTED CONCURRENCY CONTROL SCHEME[1]

Abraham Silberschatz

Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

A transaction is atomic if it can be considered, so far as other transactions are concerned, to be indivisible and instantaneous even in the case of failure. This paper examines the problem of ensuring atomicity in non-replicated distributed database systems. Our main contribution is in providing a concurrency control scheme that:

- is non-blocking; that is, the system will continue with its execution as long as there exists at least one healthy node in the system.
- allows the transactions to read uncommitted data values.
- does not use a global (central) Dependency Graph.
- does not require the use of time-stamps to order the relative execution of the various transactions in the system.

Thus our scheme is robust in the face of hardware/software failure and can be efficiently and effectively implemented in a distributed environment. Arguments in support of our scheme including correctness proofs are also presented.

[1]This research was supported in part by the Office of Naval Research under Contract N00014-80-K-0987, and in part by NSF Grant MCS-8104017.

1. Introduction

A database system in the most general sense may be simply viewed as a pair DS = <D,P>, where D is the set of the database entities, and P is the set of all the programs that may access D. A transaction in such a system is the execution of one of the programs in the set P. An important issue that arises in the design of a database system is the problem of ensuring the consistency of the database when it is accessed concurrently by a number of asynchronously running transactions. A common approach to the problem is to define a transaction as a unit that preserves consistency (e.g., it is assumed that each transaction, when executed alone, transforms a consistent state into a new consistent state), and require that the outcome of processing a set of transactions concurrently be the same as one produced by running these transactions serially in some order. A system that ensures this property is said to be serializable [1]. Associated with concurrent access to data is the problem of ensuring that for each transaction, either all of the actions associated with it are completed or none are completed. A system that ensures this property is said to be recoverable. A transaction executing in a serializable and recoverable system is said to be atomic.

In order to ensure atomicity some form of supervision must be present in the system to influence the manner in which the transactions executing the database can interact with each other. Such a supervision may be achieved through the use of a variety of methods, the most common one being a locking protocol together with some recovery scheme.

A significant amount of work has been performed in the area of locking protocols and recovery schemes for centralized and distributed environments. (For a discussion on this topic see [2]). In this paper we wish to focus our attention on the problem of locking and recovery in non-replicated distributed database systems [2-5]. Our main contribution is in providing a concurrency control scheme that:

a. is non-blocking [6]; that is, the system will continue with its execution as long as there exists at least one healthy node in the system.

b. allows the transactions to read uncommitted data values.

c. does not use a global (central) Dependency Graph.

d. does not require the use of time-stamps [7,8] to order the relative execution of the various transactions in the system.

Thus our scheme is robust in the face of hardware/software failure and can be efficiently and effectively implemented in a distributed environment.

The remainder of the paper is organized as follows. In Section 2 we present the atomic transaction model and establish some basic properties of our model. In Section 3 we develop a distributed control scheme which is independent of any particular locking scheme. In Section 4 we presented arguments in support of our proposal and discuss the question of appropriate locking

protocols in which our distributed control scheme can be effectively embedded. Finally, correctness proofs are given in the appendix.

2. An Atomic Transaction Model

A transaction is the execution of a single sequential program residing in one and only one site. The sequentiality requirement is imposed for ease of presentation; our model could be extended in a straightforward manner to handle the case where several sites may be executing simultaneously on behalf of a single transaction. A transaction may access data residing in either a single (own) site (central environment), or multiple sites (distributed environment). Thus, in a distributed environment, a transaction may be viewed as a locus of control, migrating from one site to another, carrying with it the relevent information needed to perform its designated task. At any time a transaction is in one of the following states:

active -- its initial state

partially commit -- after reaching its last statement

commit -- after "successful" completion

abort -- after discovering some "error"

fail -- after "error" taken care of

We will say that a transaction has committed/failed only if it has entered the commit/fail state. Furthermore, such a transaction will be said to have terminated.

A transaction starts its execution in the active state. It enters the partially commit state after reaching its last executable statement. It then may enter the commit state only if the serializability and robustness properties are preserved. A transaction enters the abort state after discovering that it can no longer proceed with its normal execution (e.g., hardware or logical errors). Such a transaction must be rolled back to its initial state. Once this has been accomplished the transaction enters the fail state. At this point in time the system must decide whether to try and restart this transaction (such a transaction will be considered as a new transaction), or to "kill" it.

We note that it may be necessary to rollback a transaction even if it has reached the partially-commit state (in contrast, once a transaction enters the commit state it may never be rolled back). This implies that observable external "writes" (e.g., to a terminal, printer) should only be allowed to occur after a transaction has committed. Alternatively, external writes can be allowed to occur in the midst of execution, and "appropriate" messages printed, if a rollback was performed; for ease of discussion we restrict our attention here to the former case. One way to implement such a scheme is to store any value associated with such external writes in a temporary stable storage, and to perform the actual writes only at commit time. More on this

subject later.

Since several transactions may be concurrently executing in the system, the state transition of one transaction may influence the state transition of another transaction. In order to ensure proper state transition, the system must maintain sufficient information about how the various transactions have interacted. Such information will be maintained in the form of a directed graph, called Dependency Graph (DG). The nodes of a DG are transactions which have not yet terminated, while the arcs of a DG correspond to some dependency relation among the various transactions. More precisely, a directed arc from T_i to T_j will be inserted in a DG if there exists an entity e, accessed by both T_i and T_j where at least one of them updated e, and T_j accessed it after T_i did.

Our definition of a Dependency Graph is similar in many respects to other graph models used in discussing the issues of concurrency and consistency. Dependency Graphs, however, also allow us to discuss the issue of failure.

Given a Dependency Graph, we can now precisely specify, when the state transitions of a transaction T_i may occur.

1. T_i enters commit state $\Leftrightarrow$
    a. reached last statement, and
    b. has no more fathers in the DG.
2. T_i enters abort state $\Leftrightarrow$
    a. has encountered an error, or
    b. one of its fathers in the DG has entered the abort state.
3. T_i enters fail state $\Leftrightarrow$
    a. reached abort state, and
    b. has rolled back to its initial state.

We can now state some basic properties of our scheme.

Lemma 1: If no cycles are ever formed in the Dependency Graph, then the system is serializable.

Lemma 2: If a transaction terminates, then it is executed as an atomic transaction.

There is one more issue concerning our model that needs to be addressed at this time. It concerns the problem of maintaining the Dependency Graph in a distributed environment. The observant reader has probably noticed that in our model, if a cycle is formed in a DG, then each transaction participating in the cycle must be aborted. This implies that the system must continuously search for the formation of such cycles. If the DG is distributed over a number of

different sites the detection of such cycles is not an easy task. Moreover, it requires a considerable amount of overhead.

An alternative, and one which we adopt, is to restrict our attention to those schemes which ensure that no cycles will ever be formed in a DG. Such schemes have the advantage that they can be effectively and efficiently implemented in a distributed environment. In section 4 we discuss this issue in greater detail and show how this requirement can be effectively met in a distributed environment.

3. The Distributed Control Scheme

Consider a distributed system consisting of m nodes, N_1, N_2, ..., N_m. Our task is to develop an effective mechanism to implement the dependancy graph model introduced in the previous section. In particular our mechanisms should ensure:

a. efficient implementation,

b. resiliency in case of failure.

A considerable amount of work has been done in the area of distributed implementation of atomic transactions (e.g., [2-6,8-10]). The main differences between our scheme and previous ones are:

a. Transactions are allowed to read uncommitted data values.

b. We do not use a global (central) Dependency Graph.

c. We do not require the use of time-stamps [7,8] to order the relative execution of the various transactions in the system.

We assume that the individual nodes are capable of providing both stable and volatile storage. Stable storage does not lose its contents as a result of failure, while volatile storage may lose its content as a result of failure. For a more thorough discussion on the implementation issues concerning such storage we refer the readers to [10]. For the rest of the paper we assume that all data structures described are of a volatile type. Those structures that need to be kept in stable storage will be explicitly mentioned.

The system's nodes must communicate with each other in order to ensure proper state transitions of the various transactions. Thus a node may send a message to another node via:

send <message> to <node_id>

We assume that if <node_id> is "up" then the message will be properly delivered to its destination before the next statement can be executed (this may require several lower level transmissions). If <node_id> is "down" then the message is lost (a notification of this event may be conveyed to the sender). We also require that each node process its arriving messages in the First Come First Served order. This is not a restriction since the receiver can always

internally queue these messages for later use.

For convenience, we allow a node to send the same message to all other nodes via the broadcast statement which has the form:

broadcast <message>;

The broadcast statement is semantically equivalent to a program segment consisting of N individual send statements. Thus a broadcast statement cannot be considered to be atomic. That is, if a failure occurs during its execution, some of the nodes in the system may receive the message while others will not.

The central concepts of our distributed control scheme can now be described.

1. Each node N_i maintains its own local Dependancy Graph DG_i. This graph contains information concerning only those transactions which accessed its local entities. There is no global dependency graph maintained in the system.

2. There exists a single coordinating node C whose function is to ensure the proper state transition of all the transactions in the system.

3. Our scheme assumes an unbounded growth of the Dependency Graphs. Since in such an environment a failure may be quite devastating, our scheme may be viewed as being an optimisitic one. That is, we assume that failures (abortions) occur very infrequently. In a previous paper [11] we have argued that with reasonable discipline this assumption can be effectively realized. In section 4 we briefly elaborate on this issue.

4. Our scheme uses a two-phase commit protocol:

   Phase 1 – When a transaction T reaches a state when it can be "fully" committed; that is, there is no DG_i with indegree(T) greater than zero, a message [OK T] is sent to the coordinator notifying him of the occurence of this event. When the coordinator receives this message a message [commit T] is broadcast to all nodes.

Phase 2 – The "real" commit message is broadcast. This message is piggybacked on the next "regular" message broadcast by the coordinator to all the nodes. Only when a node receives this "real" commit message is the transaction considered to be committed at this node.

We emphasize again that in contrast to previous work on two phase commit protocols we allow a transaction to access data produced by a transaction which has not yet committed (or even partially-committed).

5. The scheme is robust in the face of hardware failure. When the coordinator fails another one may be elected. Similarly, a failed node can gracefully recover. Our scheme however, does not handle the problem of network partitioning.

6. Our algorithms are quite general in the sense that they are applicable to those locking protocols which ensure serializability and deadlock-freedom. Thus our control scheme can be superimposed on any such previously developed locking protocol.

The algorithms presented in this section are driven by control messages sent and received by the coordinator and the various nodes in the system. In particular, the coordinator may issue the following control messages:

[abort T] – This message is sent to all nodes when C discovers that T must be aborted.

[commit T] – This message is sent to all nodes when C decides to commit T.

[down N_i] – This message is sent to all nodes when the coordinator (or some other node) discovers that node N_i is down.

The coordinator may receive any of the following control messages:

[wound T] – This message is sent to the coordinator by a node who discovers that transaction T may have failed.

[kill T] – This message is sent to the coordinator by a node who determines that transaction T must be aborted (i.e., T was executing in N_i and an error was discovered which prevents T from continuing with its execution.)

[OK T] – Is issued by the owner of transaction T when T reaches a state when it can be fully committed.

We now formally present the various algorithms that the coordinators and the rest of the nodes of the system must execute. The algorithms are written in a pseudo Pascal notation. In particular, if A is a set of elements of type x, then the statement

for x in A do S;

enumerates all the elements in A.

3.1. Node N_i

Each node N_i maintains the following data structures whose purpose is explained below:

- **DG_i** – a local Dependancy Graph. Each transaction in DG_i must have accessed an entity defined in N_i.
- **$Commit_i$** set – an ordered set consisting of all committed transactions in the system.
- **PC_i** set – a set consisting of all transactions which have partially committed and whose owner is N_i.
- **$Kill_i$** set – a set consisting of all transactions for which N_i has issued [kill T]. Such a message is sent when N_i discovers that T can no longer proceed with its normal course of execution.
- **OK_i** set – a set consisting of all transactions for which N_i has issued [OK T].
- **$Abort_i$** set – a set consisting of all aborted transactions.
- **$In\text{-}Out_i$** array – an array of M elements indexed by node names. Each element is a list of transaction names computed as follows:
  $In\text{-}Out_i[N_j] = \{T \mid N_i$ sent/received information concerning T to/from $N_j\}$.
  In other words, we record in $In\text{-}Out_i[N_j]$ the set of all transactions that executed in both nodes N_i and N_j.
- **$Last\text{-}commit_i$** – a variable of type transaction-id. This variable contains the name of the last transaction for which $node_i$ has received a commit message from the coordinator.

The above data structures, except DG_i, are maintained in volatile storage. The DG_i structure, however, needs to be maintained in stable storage so that in case of failure, recovery will be possible.

Let T be a transaction owned by node N_i. With each such transaction the data set:
$Before(T) = \{T' \mid T'$ is a father of T in one of the DG_is$\}$
is associated. This set is constructed while T migrates from one node to another.

When T reaches its last statement (i.e., it enters a partially commit state), the following

actions are then taken:

```
PC_i := PC_i + [T];
for T' in Commit_i do
   if T' in Before(T) then Before(T) := Before(T) - [T'];
if Before(T) = 0 then send [OK T] to coordinator;
```

Let EXTERNAL-WRITES(T) stand for "execute the external observable writes associated with T" (if there are any). Now, since failure can occur at any arbitrary point in time, such external writes cannot be immediately executed when T receives a [commit T] message. Instead, this is allowed to occur only after N_i has received both a [commit T] message followed by a regular control message (i.e., one which does not deal with recovery; see Section 3.3) from the coordinator. For this reason we have introduced the variable Last-commit$_i$. It is in this variable where we record the name of the last transaction (say T) for which node N_i received a [commit T] message. Thus whenever N_i received a regular control message it does the following:

```
if Last-commit_i ≠ '' then
   begin
      Commit_i := Commit_i + [Last-commit_i];
      EXTERNAL-WRITES (Last-commit_i);
      if Last-commit_i in PC_i then PC_i := PC_i - [Last-commit]
      delete Last-commit_i from DG_i;
      for T in PC_i do
         if Last-commit_i in Before(T) then
            begin
               Before(T) := Before(T) - [Last-commit_i];
               if Before(T) = 0 then send [OK T] to coordinator;
            end
      Last-commit_i := '';
   end
```

Thus the "actual" commit occurs at this point in time.

We present now the various algorithms that need to be executed by the node N_i in response to receiving the various control messages from the coordinator.

1. [commit T] –

   Last-commit$_i$:= T;

2. [abort T] –

```
if T in DG_i then
   begin
      Let A be the ordered set consisting of T plus all its
      descendants in DG_i.  This set is ordered in the
      reverse DG_i order.
      for T' in A do
         begin
            rollback T'
            delete T' from DG_i
            send [Kill T'] to coordinator;
            Kill_i := Kill_i + [T'];
            if T' in PC_i then PC_i := PC_i - [T'];
```

```
            end
        end
    else Abort_i := Abort_i + [T];
```

3. [down N_j] –

```
for T in In-Out_i[N_j] do
    send [wound T] to coordinator;
```

4. [new-coor T] – This message is sent by a newly elected coordinator. The value of T corresponds to the last committed transaction in the system as will be explained in Section 3.3.

```
if Last-commit_i ≠ T then
    begin
        EXTERNAL-WRITES(Last-commit_i);
        delete Last-commit_i from DG_i;
        if Last-commit_i in PC_i then PC_i := PC_i - [Last-commit_i];
        Commit_i := Commit_i + [Last-commit];
        for T' in PC_i do
            if Last-commit in Before(T') then
                begin
                    Before(T') := Before(T') - [Last-commit];
                    if Before(T') = 0 then send [OK T'] to
                                           coordinator;
                end
        Last-commit_i := T;
    end
```

When N_i receives a message from N_j to process transaction T (i.e. T has migrated from node N_j to N_i), it must first check to see whether T is in the $Abort_i$ set; if this is the case then N_i simply ignores this message. This is done in order to guard against the possibility that an [abort T] message, and the message concerning transaction T, will cross each other in some obscure way.

Finally, we note that N_i can remove T from its $In\text{-}Out_i$ array and $Kill_i$ set after the coordinator has successfully executed either [abort T], or [commit T], or [new-coor T].

3.2. Coordinator

The function of the coordinator node C is to ensure the proper state transition of all the transactions in the system. In particular, all messages concerning commitments and abortions are handled by the coordinator. The sole purpose of the coordinator is to make our scheme non-blocking; that is, to ensure that the system will continue with its execution as long as there is at least one healthy node.

The coordinator maintains two main data structures whose purpose is explained below:

- Commit set – an ordered set consisting of all committed transactions
- Abort set – a set consisting of all aborted transactions.

We present now the various algorithms that the coordinator must execute in response to

various control messages.

1. [OK T] –

```
if T not in Abort then
begin
   Commit := Commit + [T];
   broadcast [commit T];
end
```

We note that the check for determining whether T is in the Abort set is indeed required. To illustrate this, consider the case where node N_i in which T executed went down. Node N_j determines that $T \in \text{In-Out}_j[N_i]$ and sends a message to the coordinator, who then aborts T. The owner of T in the meantime may send [OK T], but T now is in the Abort set.

2. [wound T] –

```
if T not in Commit then
begin
   Abort := Abort + [T];
   broadcast [abort T];
end
```

3. [kill T] –

Same as [wound T]; we use two different commands to differentiate between the case where a node explicitly wishes to abort a transaction ([kill T]) and the case where a node simply responds to the coordinator's [down N_i] message. Such a differentiation is needed when an election of a new coordinator takes place, as will be seen in Section 3.4.

4. [down N_i] –

```
broadcast [down N_i];
```

We note that the coordinator may not decide on the abortion of a transaction. It can issue [abort T] only in response to [wound T] and [kill T] messages it receives. We do not however exclude the possibility that the coordinator may issue "broadcast [down N_i]" (.i.e, it can be viewed as the coordinator sending [down N_i] to itself). Such a situation will occur when the coordinator discovers that N_i is down.

3.3. Recovery from hardware fallure

In this section we describe the recovery actions that need to take place after a node has been down. We take the approach that if node N_i was down then this has been detected by some other node and appropriate actions have been taken (i.e., abort those transactions associated with N_i). In the following we present a scheme to ensure this.

Let N_i be a node which went down and is trying to recover. The following actions need to take place before N_i can be considered to be up.

a. find out who the current coordinator is; if no coordinator exists wait.

b. send a message to the coordinator informing him that N_i is trying to recover. It then

receives from the coordinator a copy of its Commit and Abort sets. N_i uses this copy to initialize its own $Commit_i$ and $Abort_i$ sets.

c. the following actions are then executed:

```
for T in Abort_i do
   if T in DG_i then
      begin
         rollback T;
         delete T from DG_i
      end
for T in DG_i do
   send [wound T] to coordinator;
for T in Commit_i do EXTERNAL-WRITES(T);
broadcast [up N_i]
```

After all the above have taken place, node N_i is considered to be active again in the system.

We note that DG_i was put in stable storage in order to be able to execute step (c) above. Alternatively, one could keep DG_i in volatile storage and construct a list of all the transactions that are currently active in node N_i, and keep this new list in stable storage. Step (c) above could be easily modified to deal with this change. This new scheme is simpler to implement in terms of the actual implementation of storing the information in stable storage.

3.4. Election of a new coordinator

When the coordinator node goes down, a new coordinator must be elected. There are various methods which can be used in electing a coordinator [4,12] and for our scheme any one of them will do. We do however require that each node that is up will participate in the election. In addition, each such node will freeze its activities until it either receives a [new-coor T] message, or a new election is declared. Before responding to messages from the possibly newly elected coordinator each such node will be required to process all the control messages it has received thus far. This is done to ensure that the $Commit_i$ sets are in a consistent state.

Suppose that a new coordinator has been elected, say C. When C is notified about his new job he must execute the following:

a. send a message to all nodes informing them that he has been elected as a possible coordinator.

b. get from each node N_i that is up the sets $Commit_i$, $Kill_i$, $Abort_i$, OK_i and the variable $Last\text{-}commit_i$. C constructs its own Commit and Abort sets from the information it has obtained.

c. Let Commit be a newly constructed set:

```
for T in Commit do broadcast [abort T].
```

We note that there is no need for the coordinator to be concerned with broadcasting [Abort T] messages for transactions that depend on those in the set F, since each node N_i's algorithm for [Abort T] calls for it to rollback the dependent transactions that it is aware of, based on its DG_i.

d. Let Kill be the set consisting of all transactions appearing in all the $Kill_i$ sets the coordinator has received in step (b).

```
for T in Kill do
   if T not in Commit then
      begin
         Abort:= Abort + [T]
         broadcast [Abort T]
      end
```

e. Let D be the set of nodes that are currently down.

```
for N in D do broadcast [down N];
```

f. C waits until all nodes that are "up" have sent them their appropriate [wound T] messages. For each [wound T] message an appropriate [abort T] message is sent. Note that any node may fail in the midst of sending its [wound T] messages. Such a node must be added to the set D and steps (d) and (e) must be repeated.

g. if all Last-commit$_i$ = '' (from step (b)), then goto step (g). Otherwise, let T be the transaction such that there exists Last-commit$_i$ = T and T not in the newly constructed commit set.

broadcast [new-coor T]

It can be easily shown that there exists such a unique transaction T.

h. Let OK be the set of all transactions appearing in all the OK_i sets the coordinator received in step (b).

```
for T in OK do
   begin
      if T not in Abort and T not in Commit then
         begin
            Commit:= Commit + [T]
            broadcast [Commit T]
         end
   end
```

3.5. Remarks

There are three additional issues that need to be discussed concerning our distributed control scheme. The first one concerns the problem of maintaining the database entities, the second one concerns the problem of the unbounded growth of the Commit and Abort sets, while the last one is concerned with the issue of proving the correctness of the proposed scheme. Below we deal with the first two issues. Arguments in support of the correctness of our scheme are presented in the appendix.

3.5.1. The Maintenance of the Database Entities

Our distributed control scheme implicitly assumed that when the coordinator issues a [commit T] message, no hardware failure can effect this commit. That is, all the changes made in the database as a result of the execution of T can always be recovered. There are a variety of techniques for ensuring this property. For example, requiring all data items to be kept in stable storage. Another more economical method is to require that before a transaction T leaves a node N for the last time, N must store all changes made by T in stable storage. For sake of brevity we

do not elaborate on this subject in this paper.

3.5.2. The Commit and Abort Sets

One point which we have not discussed thus far is the unbounded growth of the Commit and Abort sets.

i. It can be easily shown that it is safe to remove a transaction T from the Commit sets if no node, which is either up or down, maintains a DG_i which includes T. This observation can be used in a variety of ways to remove transactions from the Commit sets; we do not elaborate on this subject any further in this paper.

ii. It can be shown that it is safe to remove a transaction T from the Abort sets if the following two conditions hold:

   a. there exists a coordinator which did not fail in the midst of broadcasting [abort T],

   b. all messages concerning transaction T have been processed in the system.

   It is easy to determine when condition (a) holds. However, it is a little bit more difficult to handle condition (b). One possibility is to use a time out mechanism; yet another mechanism is to use a cancellation message. Again for the sake of brevity we do not elaborate on this subject any further in this paper.

4. Discussions

It should be clear that if a DG can unboundedly grow, failure may be quite devastating. This is due to the fact that a transaction which is rolled back must first enter the abort state and thus by our algorithm, each of its successors in the DG must also enter the abort state and be rolled back. (This situation is commonly known as cascading rollbacks.) On the other hand, by allowing the DG to unboundedly grow one may obtain a greater amount of parallelism.

In order to control the growth of a Dependency Graph some form of supervision must be present in the system to influence the manner in which the transactions executing the database can interact with each other. Such a supervision may be achieved through the use of various concurrency control schemes, the most common one being locking protocols.

The two-phase locking protocol [1] is the most "popular" protocol to date. This protocol allows a transaction to lock as many data items as it needs. However as soon as it unlocks one of these items no more locking is allowed. The reason the two-phase scheme is so popular is because

it deals effectively with cascading rollbacks. However, in order to completely eliminate cascading rollbacks, a strict two-phase scheme must be used [13]. Such a scheme requires that unlocking may only be performed after the transaction has reached its last statement. Thus in terms of the Dependency Graph model the strict two-phase scheme ensures that the DG will be of minimal size.

Recently, a number of new non-two-phase protocols have been proposed which ensure serializability [14-22]. All of these protocols rely on a priori information as to how the database entities are organized. The advantage of the non-two-phase protocols is that they enhance parallelism. Unfortunately, they do not guard against cascading rollbacks. Thus in terms of the Dependency Graph model these protocols allow the DG to unboundedly grow.

What we have pointed out is that there are at least two conflicting criterions in selecting a locking protocol. One is to guard against cascading rollbacks while the other is to increase parallelism (i.e., minimize locking time). If a system can ensure that transactions failure will occur only relatively infrequently, then cascading rollbacks may be tolerated, otherwise, it should be guarded against. In a previous paper [11] we have argued that with some reasonable discipline, "abortion" rates can be considerably reduced. In particular we have noted that there are three events which will cause a transaction to enter an abort state; namely, cycle formation in a DG, hardware and software errors, and the transaction cancelling itself.

We have already seen that cycle formation in the DG may be eliminated via the use of locking protocols which ensure serializability. Unfortunately, the act of locking may create a new kind of cycle commonly known as deadlock. Deadlocks must be resolved via the use of rollbacks, which in our model may result in cascading rollbacks. Furthermore, deadlock detection in a distributed environment is quite expensive. Thus in order to minimize the overhead and reduce cascading rollbacks, one must choose locking protocols that assure both serializability and deadlock freedom. The two-phase protocol does not ensure freedom from deadlocks. This deficiency can be remedied through the use of some prevention algorithm, such as hierarchical ordering of the entities in the database. Most of the non-two-phase protocols developed thus far ensure freedom from deadlock. Those protocols that are not deadlock-free enjoy the property that deadlocks can be effectively handled without causing cascading rollbacks [23].

Obviously, hardware failure cannot be completely eliminated. There are various degrees of hardware failures, each of which may result in different recovery actions. Some of these errors may be corrected by simply repeating a particular action (e.g. rewrite a page on a disk). Other errors however, may require the abortion of a transaction (and its successors in the DG). In recent years we have seen a significant improvement in terms of hardware reliability. A case in point is the Tandem Computer which improves reliability by duplicating every major hardware component [24]. In such an environment hardware failure (coupled with software for recovery) is

a rare event. Since the cost of hardware is going down, such a technology is cost effective. We thus expect in the future that most hardware will use some duplication (and even triplication) in order to improve reliability.

In [11] we have argued that a database system should not allow the user to cancel a transaction in mid-execution. We do not have any objection to such cancellation if the data on which the transaction is operating on, is either private, or is explicitly locked by the owner of the transaction. If the data is private, the abortion of the transaction will not affect any other transaction. Explicit locking will result in the notification of the system that the transaction may be aborted in the midst of execution. The system then may decide to employ a strict two-phase locking for this particular transaction so that in case of abortion no cascading rollbacks will result. In all other cases, cancellation of a transaction should not be permitted. An exception to this is a transaction which does not modify any of the database entities. Such a transaction can always be aborted without directly effecting other transactions.

Finally, we note that even in case of failure, it may be possible to improve performance by using partial rollbacks. In [25] we have described such a scheme for the two-phase locking protocol. We are currently extending our results to other concurrency control schemes, including non-two-phase protocols and multiversion protocols.

5. Conclusion

The problem of ensuring the atomicity of each database transaction was considered. We have focused our attention on non-replicated distributed database systems. We have developed a distributed control scheme which can be superimposed on any of the previously developed "safe" locking protocols. We have shown that our scheme is robust in the face of hardware/software failure and can be effectively and efficiently implemented in a distributed environment. Finally, we have pointed out that with reasonable discipline the number of rollbacks required in a system can be considerably reduced. Thus contrary to common practice non-two-phase protocols could be effectively used in guaranteeing atomicity.

Lastly, we note that if one combines our scheme with a locking protocol that assures serializability, one can modify the Dependency Graph model to reduce the number of dependencies in the system. In particular, we can insert a directed arc from T_i to T_j in a DG only if both transactions accessed a common entity e, and T_i wrote onto e and T_j read the value written by T_i. This modification is possible since the DG is only used for commitment decisions.

REFERENCES

1. Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. The notions of consistency and predicate locks in a database system. CACM 10, 11 (Nov. 1976), 624-723.

2. Gray, J., Notes on database operating systems. Operating Systems, Lecture Notes in Computer Science: Volume 60, Springer Verlag, 1978.

3. Rosenkrantz, D.J., Stearns, R.E., and Lewis, P.M. System level concurrency control for distributed database systems. ACM Transactions on Database Systems 3, 2 (June 1978), 178-198.

4. Menasce, D.A., Popek, G.J., and Muntz, R.R. A locking protocol for resource coordination in distributed databases. ACM Transactions on Database Systems 5, 2 (June 1980), 103-138.

5. Bayer, R., Elhardt, K., Heller, H., and Reiser, A. Distributed concurrency control in database systems. Proc. International Conference on Very Large Data Bases (Oct. 1980), 275-284.

6. Skeen, D. Nonblocking commit protocols. Proc. ACM-SIGMOD International Conference on Management of Data (April 1981), 133-142.

7. Lamport, L. Time, clocks, and the ordering of events in a distributed system, CACM 21, 7 (July 1978), 558-565.

8. Reed, D. Naming and synchronization in a decentralized computer system. Technical Report, MIT, Sept. 1978.

9. Hammer, M., and Shipman, D. Reliability mechanisms for SDD-1: A system for distributed databases." Computer Corporation of America, Cambridge, Mass., (July 1979).

10. Lampson, B., and Sturgis, H. Crash recovery in a distributed data storage system, CACM, (to appear).

11. Silberschatz, A. A case for non-two-phase locking protocols that assure atomicity. IEEE Transactions on Software Engineering, (to appear).

12. Garcia-Molina, H. Election in a distributed computing system. Technical Report, Princeton University, Dec. 1980.

13. Stearns, R.E., Lewis, P.M., and Rosenkrantz, D.J. Concurrency control for database systems. Proc. IEEE Symposium on Foundations of Computer Science (Oct. 1976), 19-32.

14. Silberschatz, A., and Kedem, Z. Consistency in hierarchical database system. JACM 27, 1 (Jan. 1980), 72-80.

15. Kedem, Z., and Silberschatz, A. Controlling concurrency using locking protocols. Proc. 20th IEEE Symposium on Foundations of Computer Science (Oct. 1979), 274-285.

16. Yannakakis, M., Papadimitriou, C.H., and Kung, H.T., Locking policy: safety and freedom from deadlock. Proc. 20th IEEE Symposium on Foundation of Computer Science (Oct. 1979), 286-297.

17. Kedem, Z., and Silberschatz, A. Non-two-phase locking protocols with shared and exclusive locks. Proc. Sixth International Conference on Very Large Data Base (Oct. 1980), 309-317.

18. Kedem, Z., and Silberschatz, A. A characterization of database graphs admitting a simple locking protocol. Acta Informatica 16, 1 (1981), 1-13.

19. Fussell, D., Kedem, Z., and Silberschatz, A., A Theory of correct locking protocols for database systems, Proc. Seventh International Conference on Very Large Data Bases (Sept. 1981), 112-124.

20. Silberschatz, A., and Kedem, Z. A family of locking protocols for database graphs modeled by directed graphs. IEEE Transactions on Software Engineering 8, 6 (Nov. 1982), 558-562.

21. Yannakakis, M.A. A theory of safe locking policies in database systems. JACM 29, 3 (July 1982), 718-740.

22. Kedem, Z., and Silberschatz, A. Locking protocols: from exclusive to shared locks. JACM (to appear).

23. Kedem, Z., Mohan, C., and Silberschatz, A. An Efficient Deadlock Removal Scheme for Non-Two Phase Locking Protocols, Eighth International Conference on Very Large Data Bases, (September 1982).

24. TANDEM, Non Stop System Description, Tandem Computers, Inc., Cupertino, CA.

25. Fussell, D., Kedem, Z., and Silberschatz, A. Deadlock removal using partial rollback in database systems, Proc. ACM-SIGMOD International Conference on Management of Data (April 1981), 65-73.

6. Appendix: The Correctness of Our Scheme

We present now arguments in support of the correctness of our scheme. Since no cycles can be formed in the Dependency Graph, the serializability property is ensured. Thus we only need to show that our scheme ensures robustness. In order to do so we need to prove that the following three properties always hold.

1. Under no circumstances will there be in the system a transaction T and two nodes N_i and N_j such that N_i executed the external observable writes associated with T while N_j rolled T back to initial state.
2. Under no circumstances will there be two transactions T_1 and T_2 such that T_1 has aborted, T_2 has committed and T_2 depends on T_1.
3. Every transaction in the system either commits (i.e., all its external observable writes are eventually executed), or fails (i.e., is rolled back to its initial state.)

For brevity some of the details of the proofs are omitted.

Claim 1: If the coordinator issued [abort T], then for all nodes N_i the following holds:

- $T \notin Commit_i$
- $Last\text{-}commit_i \neq T$.

Proof: The proof follows from the following two facts:

1. for each node N_i, $Commit_i \cup Last\text{-}commit_i \subseteq Commit$.
2. the coordinator broadcasts an [abort T] statement only if T is not in the Commit set.

Claim 2: If the coordinator issued [abort T], then neither [commit T] nor [new-coor T] messages will ever be issued.

Proof: Assume by contradiction that a coordinator has issued [abort T], and that later on a [commit T]/[new-coor T] message was issued. We claim that between the time [abort T] and [commit T]/[new-coor T] were issued the coordinator (say C1) must have failed. Moreover, C1 must have failed in the midst of broadcasting [abort T]. Indeed, when the coordinator issues [abort T] it adds T to its Abort set. Thus if a [OK T] message was on its way to the coordinator, by our algorithm it will be simply ignored. Also, since each N_i adds T to its own $Abort_i$ set, from that point all non-control messages concerning T will be ignored. Thus C1 could not have received a [OK T] message. From Claim 1 we know that the newly elected coordinator (say C2) could not have issued [new-coor T]. Furthermore, C1 must have failed in the midst of broadcasting [abort T]; otherwise, by our algorithm the owner of T would not have issued (or reissued) the message [OK T]. Thus we need only consider the case where the coordinator C1 failed in the midst of broadcasting [abort T] and the owner of T issued, at some point in time later, the message [OK T]. Three subcases need to be analyzed.

i. C1 issued [abort T] as result of a [kill T] message from node N_k. If when C2 was

elected N_k was still up then by our algorithm N_k will send to C2 its $Kill_k$ set and thus C2 will issue again [abort T]. Thus we are left with the case where N_k went down before C2 was successfully elected. Consider the chain $<N_k,N_{k1},...,N_{kq}>$ consisting of all the nodes through which T has migrated up to the point in time when C1 failed (note that the owner of T must be in this chain). When C2 is elected N_k is still down (it may be trying to recover, but since no coordinator exists, it is still considered to be down). If none of the nodes in the chain is up when C2 was elected then it implies that the owner of T is also down. Thus no [OK T] message can be issued. Furthermore, as soon as one of these nodes tries to recover, by our algorithm, it will issue [wound T]. Since T is not in the Commit set it will be aborted. If on the other hand, when C2 was elected, one of the nodes in the chain is up, we have the following. First observe that each node in this chain must have T in its own In-Out array. (T is removed from the In-Out array only after the coordinator has successfully completed broadcasting [abort T]). Second, let N_{ki} be a node such that N_{ki} is down and N_{ki+1} is up. When C2 is elected it discovers that N_{ki} is down and thus broadcasts [down N_{ki}]. In response N_{ki+1} will issue [wound T]. Since T is not in the Commit set, C2 will abort it.

ii. C1 issued [abort T] as result of a [wound T] message from node N_k, and N_k sent [wound T] to C1 as result of C1 broadcasting [down N_i], and T is in $In\text{-}Out_k(N_i)$. If N_i recovered between the time [down N_i] was received by N_k and the time N_k issued [wound T], then by our algorithm C1 will again issue [abort T]. If N_i does not recover this implies that when C2 is elected it will discover that N_i is still down (N_i may be trying to recover). Thus C2 will issue [down N_i]. If N_k is up then it will reissue [wound T]. If N_k is down arguments similar to those present in case (i) above can be repeated.

iii. C1 issued [abort T] as a result of a [wound T] message from node N_k, and N_k sent [wound T] to C1 as part of its recovery algorithm (section 3.4). Since C1 failed in the

midst of broadcasting [abort T] by our algorithm, when C2 is elected N_k is still considered to be down. Thus when C2 is elected it discovers that N_k is down and it broadcasts [down N_k]. Since no coordinator has successfully broadcast [abort T], it follows that T has not been removed from any In-Out array. Thus we can repeat the same arguments as in case (i).

Since [new-coor T] can be issued only after [commit T] was issued, the result follows.

Claim 3: If the coordinator does not fail in the midst of broadcasting either [commit T] or [new-coor T], then no [abort T] messages will ever be issued.

Proof: Suppose that the coordinator has not failed in the midst of broadcasting either [commit T] or [new-coor T]. Let $X=\{N_1,N_2,...,N_q\}$ be the set of nodes that were "up" immediately after the coordinator has completed its broadcast. Clearly each N_i in X will have Last-commit$_i$ = T. We claim that from this point on, every node that will be up will have T in either its Commit set or in its Last-commit variable. Indeed, if the coordinator fails, the new elected coordinator will have T in its commit set. Now since every node that recovers (say N_k) constructs its Commit$_k$ set out of the commit set it receives from the coordinator, the subclaim has been substantiated. Thus every coordinator will have T in its commit set and by our algorithm no [abort T] messages will ever be issued.

We note that if the coordinator does fail in the midst of broadcasting either [commit T] or [new-coor T], it is possible that an [abort T] message will indeed be issued. To illustrate this, consider the case where the coordinator has failed just after it has sent its first [commit T] message to node N_i. Suppose that N_i and the owner of T have failed. When a new coordinator is elected no record is available to indicate that T has "potentially" committed. Now since the owner of T is down, by our algorithm the new coordinator will abort T. It is for this reason that the external writes of a transaction cannot be immediately executed when a [commit T] message is received.

Claim 4: The procedure EXTERNAL-WRITES(T) can be executed only if there exists a coordinator which did not fail in the midst of broadcasting either [commit T] or [new-coor T].

Claim 5: Property 1 always holds in the system.

Proof: Assume by contradiction that claim 5 is false. Let T be a transaction such that N_i has executed EXTERNAL-WRITES(T) and N_j has rolled T back. Since N_i executed EXTERNAL-WRITES(T), by Claim 4 some coordinator did not fail in the midst of broadcasting either [commit T] or [new-coor T]. By Claim 2, [abort T] was sent after the coordinator (mentioned above) had broadcast either [commit T] or [new-coor T], which contradicts Claim 3.

Claim 6: Property 2 always holds in the system.

Proof: If T_2 depends on T_1 then by our algorithm T_1 will be added to the set Before(T_2). Moreover this will occur prior to T_2 entering the partial-commit state. Now T_1 is removed from the set Before(T_2) if and only if T_1 has "fully" committed. Since an [OK T_2] message can be sent only if Before(T_2) = 0, our result has been proven.

Claim 7: Property 3 holds in the system provided that the coordinator does not fail infinitely often.

Proof: From our previous claims (and their proofs) it is clear that we need only consider the case where the coordinator fails in the midst of broadcasting [abort T], [commit T], [new-coor T].

Case 1: From the proof of claim 2, we know that in the case of [abort T], the newly elected coordinator will reissue [abort T]. From our hypothesis it follows that there will be a point in time when the coordinator successfully completes broadcasting [abort T]. Thus each node that is "up" will rollback T. Furthermore, every node that recovers will also rollback T.

Case 2: From claim 3 we know that if the coordinator fails in the midst of broadcasting [commit T], [new-coor T] then either the new coordinator will issue [abort T] and then we have case 1, or it will reissue [new-coor T]. Thus for each such T either an [abort T] message will be issued, or, eventually [new-coor T] will be successfully broadcast. In the latter case, from claim 3, no [abort T] messages will ever be issued. Moreover, from that point on T will be in the commit set of all nodes. From our hypothesis it follows that the external writes of T will be eventually executed.