CORRECTNESS OF A DISTRIBUTED TRANSACTION SYSTEM

Krithivasan Ramamritham

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 81-32

November 1981

Revised November 1982

This work was supported by the National Science Foundation under grant MCS 82-02586.

ABSTRACT

A distributed transaction system manages information that is dispersed over a number of storage devices. This paper deals with an experimental transaction system designed to satisfy real-time constraints through distributed control of the executions of transactions. Of interest is the correctness of the algorithm for distributed control. Demonstrating the correctness involves showing that the algorithm guarantees the consistency of distributed data, and equally importantly, that every transaction will eventually terminate. Proof of consistency is based on the notion of serializability of transactions while proof of termination is based on the conflict resolution and failure recovery strategies employed by the transaction system.

Note for the Printer

The symbol ∀ stands for "for all" and should be printed as ∀.

The symbol ∃ stands for "there exists" and should be printed as ∃.

The symbol ~∃ stands for "there does not exist"
and should be printed as ∄.

The symbol ∈ stands for "belongs to" and should be printed as ∈.

The symbol /\ stands for "and" and should be printed as ∧.

The sybbol V stands for "or" and should be printed as ∨.

The symbol ¬ stands for "not" and should be printed as ¬.

The symbol [] should be printed as □.

The symbol <> should be printed as ◇.

1. INTRODUCTION

A transaction system manages information that is dispersed over a number of storage devices. Examples of such systems are airline reservation systems, information retrieval systems and automated offices. Some of the requirements of transaction systems are:

- consistency - the transactions should be processed in a such a way that information in the system is always consistent.
- quick response - typically a transaction is initiated by a user at a terminal and hence the response time should be short, and
- robustness - the system should be able to complete a transaction even in the event of failure of storage or processing elements.

Systems which satisfy these requirements are in general termed distributed database management systems [5, 9, 11]. In these systems, responsiveness and robustness are achieved through the distribution of data and by maintaining duplicates of parts of the data. Access to distributed data is also improved through the distribution of various aspects of controlling the data. Different algorithms have been proposed for achieving distributed concurrency control [2, 8, 12]. In this paper we will concern ourselves with the algorithm that has been implemented as part of the Delta system [5]. This algorithm differs from others in that only timestamps of individual transactions are utilized for achieving distributed control.

An important function of any concurrency controller is to ensure that

transactions that are permitted to execute will maintain the consistency of distributed data. Another equally important function is to guarantee that a transaction once initiated will eventually terminate. Our work is directed towards demonstrating that Delta's concurrency control mechanism, called distributed executive, performs these two functions correctly. We use temporal logic [7] to formally argue about the correctness of the Delta system.

To prove that Delta's distributed executive maintains the consistency of distributed data we prove that transactions in Delta are well-formed, two-phase, and legal in the sense of [3] and hence are serializable. According to the notion of serializability, a distributed executive is guaranteed to maintain the consistency of distributed data if every set of transactions, allowed to run concurrently, is equivalent to some serial execution of transactions in that set. Two sets of transactions are equivalent if the resulting database state is the same for executions of both sets of transactions, when started at the same initial state. Since serializability is a sufficient condition for consistency, Delta's distributed executive maintains the consistency of data.

Termination of transactions is of significance in transaction systems since it relates to the responsiveness of the system. Heretofore not much attention has been paid to this "liveness" property of transaction systems. It is in general difficult to guarantee termination of transactions in the face of failure of storage or processing elements. In a distributed system such as Delta, transactions may not be able to terminate due to a number of reasons,

predominant among them being

- two or more transactions are deadlocked over access to a set of common data items.
- transactions are made to wait indefinitely for the resolution of a conflict.
- processing or storage elements needed for the execution of the transactions have failed.

The conflict resolution strategy used in Delta favors a transaction that was started earlier and thus prevents deadlocks. Since there are only a finite number of transactions that could have been started before any waiting transaction, the strategy also avoids situations where a transaction waits indefinitely. Failure recovery strategies of Delta make termination possible even in the event of failures. We approach the problem of demonstrating termination in the Delta system through successive relaxation of the assumptions concerning the nature of transactions and the functioning of the system itself. As we shall see, the final assumptions, relaxing any one of which cannot guarantee termination, indicate those system components whose failure may affect termination.

Some issues concerning the correctness of distributed control have been dealt with earlier. Thomas [12] provides plausibility arguments for the correctness of his majority consensus algorithm. He focuses on the consistency of data and deadlock among the transactions. Proof of correctness of the SDD1 concurrency control mechanism with respect to consistency requirements is reported in [2]. Rosenkrantz et al. [8] are concerned about consistency and termination properties of their concurrency control mechanisms but do not include failure of storage

elements in their discussion. Besides being concerned with a concurrency control mechanism that is conceptually different from the ones mentioned above we are interested in both consistency and termination properties of transactions in Delta even in the presence of failures.

We begin by outlining the salient features of the distributed executive of the Delta system. Only those aspects relevant to our discussion of correctness will be stressed. (A complete description of the distributed executive appears in [5].) Temporal logic and auxiliary predicates on the system states are introduced in section three. Section four is devoted to a proof of correctness of Delta's distributed executive. The implications of the proof on the implementation of the system are discussed in the concluding section.

2. THE DELTA SYSTEM

Delta is an experimental distributed transaction system built within the framework of INRIA'S Project Sirius. It allows users to concurrently access data files in physically dispersed storage devices. Agents such as human users, sensors, etc., access the system by activating transactions. A transaction consists of a finite number of elementary actions that manipulate the data objects. Create, Read, Write and Destroy are typical actions. Agents view transactions, delimited by BEGIN and END commands, as atomic. Hence it is the responsibility of Delta's distributed executive to ensure atomicity of transactions in the face of concurrent execution of transactions. Actions corresponding to a transaction are fired by the master process controlling that transaction, to be executed by processes referred to as slaves. Data objects are partitioned and replicated to survive crashes of storage elements and a particular partition is associated with a specific slave.

Underlying the Delta system is a communication subsystem which guarantees reliable communication between the components of the system. Masters are arranged along a virtual ring with each master having a unique predecessor and a unique successor. We will not go into the details of the virtual ring beyond noting that it survives failures of both processing elements and components of the communication subsystem. A particular message called the control token circulates along the virtual ring. When accessed by a master, this token delivers unique integers called tickets. A master uses these tickets to timestamp a newly initiated transaction. All actions invoked for a particular

transaction are identified through the use of the timestamp for that transaction.

2.1 Execution of transactions:

An agent in Delta initiates a transaction by submitting it to a master. That master takes charge of executing the transaction and invokes slaves to execute the actions corresponding to that transaction. These actions (Read, Write, Create, or Destroy) take place within the execution context of the invoking transaction. To simplify our discussion here, we will assume that a master invokes a slave to execute a set of actions for a transaction through a single request. Before performing the actions, slaves lock the data items needed for completing the actions. However, all modifications are done only on copies of the data involved and are made permanent only when instructed by the master at the end of the transaction.

Two transactions conflict at a slave when one is trying to modify a data item accessed by another. When conflicts occur at different slaves, they are resolved identically at all such slaves. (If t1 and t2 are the timestamps of transactions T1 and T2 respectively, and t1 < t2, then T2 is said to be younger than T1 and T1 is said to be older than T2.)

1) When a younger transaction is received by a slave and it conflicts with an older transaction, the younger transaction waits for the older one to complete.

2) When an older transaction is received by a slave and it conflicts with a younger transaction whose actions are in progress, then the younger transaction is rolled back or aborted.

To rollback or abort is a system-wide decision. We shall see the

implications of each during the discussion of termination. If a transaction is aborted by a slave, the master aborts all actions for the transaction at all slaves. In the case of a rollback, the slave concerned waits until conflicting older transactions have committed and then restarts the action. Other slaves are not affected. This conflict resolution strategy is similar to the "wound-wait" protocol discussed in [8].

To make the results of the actions permanent, the master, on encountering the END statement of a transaction, initiates a two-phase commit protocol.

> Phase-1: A master sends "prepare to commit" messages to its slaves. If the actions for that transaction have been completed and the changes, if any, can be locally committed, then a slave acknowledges by indicating its readiness to commit. Otherwise a slave indicates its need to abort. The first phase ends when every slave involved in a transaction has acknowledged the prepare to commit message.
>
> Phase-2: If one or more slaves have indicated their need to abort then all slaves are informed by the master to abort. On the other hand, if all slaves are ready to commit, then the slaves are informed by the master to commit, in which case, the slaves make the changes permanent. In either case, once slaves receive messages from the master to commit or abort, locks set on local data items are removed and the context of the transaction destroyed.

After a master decides to commit a transaction, it informs the agent for that transaction that the transaction is completed. Otherwise, the master restarts the transaction with the same timestamp.

2.2 Execution of transactions in spite of failures:

Now we briefly describe what happens in the Delta system under

different kinds of failures. Some of the solutions are simplified versions of those described in [5].

To make recovery possible in spite of failure of slaves, the Delta system is designed with enough redundancy such that when a failed slave is eventually restarted, it will be able to restore its storage elements to be consistent with the rest of the system. To facilitate recovery in the event of failure of a master, the list of slaves involved in a transaction is passed to the slaves along with the prepare to commit message. Below we examine the nature of the failures possible and how the system responds to their occurrences.

1. If a master fails before the prepare to commit message is sent to all slaves, another master recognizes this fact through the virtual ring mechanism. This master aborts the transaction controlled by the failed master and restarts it with the same timestamp.

2. If a master fails after prepare to commit messages have been sent to all slaves, then slaves communicate to decide on the course of action.
   a. If at least one slave has committed, then all commit.
   b. If all have indicated their readiness to commit, then all commit.
   c. If at least one slave has aborted, then all abort.
   d. If none of the above cases apply, then the transaction is aborted and resubmitted by another master with the same timestamp.

If some slaves have failed and if the slaves that are up have neither committed nor aborted, then they wait for the failed slaves to restart and then decide based on the above criteria.

Below we present abstract code depicting the functioning of masters and slaves, using an Ada-like syntax. (The statement labels in the code will be used in the proof.)

```
     p: master PROCESS[G:agent,SA:set of actions];
     t : timestamp;
Lp0: BEGIN
     t:=ticket obtained from virtual ring;
     Restart:
Lp1: FOR EACH slave required to execute actions SA
          LOOP send "request actions for t" message END LOOP;
Lp2: FOR EACH slave required to execute actions SA
          LOOP send "prepare to commit t" message END LOOP;
Lp3: Wait until either "ready to commit t" or "need to abort t"
          message is received from all slaves;
Lp4: IF any slave needs to abort
      THEN
          BEGIN
           Send "abort t" messages to all slaves;
           GO TO restart;
          END;
      ELSE
          BEGIN
Lp5:       Send "commit t" messages to all slaves;
Lp6:       Send "t completed" message to g;
          END;
     END;
     END master;
```

```
      c: slave PROCESS;
      BEGIN
      LOOP
      SELECT
      WHEN slave receives "request actions for t" message DO
Lc0:    IF t does not conflict with any transaction in progress
         THEN BEGIN
Lc1:          lock data needed for t;
Lc2:          access data;
              END
         ELSE BEGIN
Lc3:      IF t<timestamp of conflicting transaction t1
          THEN BEGIN
                t1 := timestamp of conflicting transaction;
                IF rollback is the conflict resolution strategy
                THEN BEGIN
Lc4:                 unlock all data entities locked by t1;
Lc5:                 let t1 wait;
                     END
                ELSE
Lc6:                 need-to-abort(t1,c) := true;
                END
           ELSE
Lc7:         let t wait;
      OR
       WHEN  no conflict exists for the oldest waiting transaction DO
        BEGIN
Lc8:    t := oldest waiting transaction;
Lc9:    lock data needed for t;
Lc10:   access data;
        END;
      OR
       WHEN slave receives "prepare to commit t" message DO
Lc11:  IF need-to-abort(t,c)
        THEN send "need to abort t" message to master of t
        ELSE BEGIN
Lc12:        Wait until all actions for t have terminated;
Lc13:        Send "ready to commit t" message to master of t;
             END;
      OR
       WHEN slave receives "abort t" message DO
        BEGIN
Lc14:   Unlock all data entities locked for t;
Lc15:   Destroy the context for t;
        END;
      OR
       WHEN slave receives "Commit t" message DO
        BEGIN
Lc16:    Make changes permanent;
Lc17:    Unlock all data entities locked for t;
Lc18:    Destroy the context for t;
        END;
      END LOOP;
      END slave;
```

3. PRELIMINARIES TO THE PROOF OF CORRECTNESS

To provide rigor to the proof of correctness, the formalism of temporal logic is employed. Hence we start with an introduction to temporal logic and its operators and then proceed to introduce certain predicates to refer to various stages of execution of masters and slaves. We end this section with precise statements for the assumptions made about Delta.

3.1 Temporal logic

Pnueli first applied temporal logic for reasoning about safety and liveness properties of concurrent programs [7, 4]. It has since been used for proving properties of concurrent programs [1, 6] and for the specification of protocols for distributed systems [10]. Following along those lines, concurrency is modeled by a nondeterministic interleaving of computations of individual processes. Each computation changes the system state which consists of values assigned to program variables and the instruction pointer of each of the processes. Using temporal logic operators, one can specify and reason about the properties of the sequence of states that results from the execution of the concurrent processes.

Since temporal logic is an extension of predicate calculus, a temporal logic statement can involve the usual logical operators V (or), /\ (and), ¬ (not) and => (implication) besides the temporal operators [], <> and UNTIL. (In what follows, P and Q are arbitrary assertions.) The operator [] is pronounced "always". []P states that P is true now

and will remain true throughout the future. The operator <> is pronounced "eventually" and is the dual of [] in that

<>P IFF ¬[]¬P.

Thus, <>P if P is true now or will be true sometime in the future. A requirement such as "every request will be serviced" can be specified as

[]("request for service exists" => <>"request serviced").

The operator UNTIL has the following interpretation:

(P UNTIL Q) IFF P will be true as long as Q is false.

(The truth value of P once Q becomes true is not indicated by UNTIL.) The UNTIL operator is typically used for expressing temporal orderings. For example, the fact that a service can not be provided until there is a request for that service can be stated as

¬("request serviced") UNTIL ("request for service exists")

The semantics of [] and <> are identical to those of the corresponding linear time logic operators of [4] whereas UNTIL is related to Lamport's binary [] operator (read AS LONG AS) in the following manner:

(P UNTIL Q) = (¬Q[]P)

In addition, we use the following derived operator.

(P ONLYAFTER Q) = (¬P UNTIL Q)

P can become true only after Q does.

Given below is a list of the theorems of the so called "linear time" temporal logic [4] that will be employed in the proof of correctness.

T-1 : []($P \vee Q$) => ([]P $\vee$ <>Q)

[]($P \wedge Q$) => ([]P $\wedge$ []Q)

T-2 : [](P => <>Q) $\wedge$ [](Q => <>R)

=> [](P => <>R)

T-3 : []P /\ <>Q => <>(P /\ Q)

T-4 : (P UNTIL Q) /\ []¬Q => []P

T-5 : ((P ONLYAFTER []Q) /\ P) => []Q

T-6 : (P ONLYAFTER Q) /\ (Q ONLYAFTER R)

=> (P ONLYAFTER R)

T-7 : (P ONLYAFTER Q) /\ [](R => P) /\ [](Q => S)

=> (R ONLYAFTER S)

T-8 : If T is a theorem then []T.

These theorems follow from the definition of the operators.

We mentioned earlier that the state of a system is made up of the values assigned to program variables and the position of the instruction pointer of individual processes. Here we introduce certain auxiliary predicates to deal with the latter.

3.2 Definition of predicates on the position of instruction pointers:

Given a statement S that is executed by some process,

at(S) IFF control of that process is at the beginning of S.

in(S) IFF control is within S.

after(S) IFF control is immediately following S.

These three predicates are mutually exclusive and become true in the above order. The formal definition of the language construct corresponding to S would specify how these internal control points are affected by the execution of S.

S-1: Given a statement sequence S;T, where S and T are executable statements then control is at the beginning of T if and only if

it is after statement S.

at(T) IFF after(S)

S-2: Assuming that the underlying scheduler of processes is fair, if control is at the beginning of a statement then control will eventually be within the statement, i.e.,

at(S) => <>in(S)

and if the statement is known to terminate, then control will eventually reach the end of the statement.

at(S) => <>after(S)

S-3: If S is

IF p THEN S1 ELSE S2

then

at(S) => [](at(S1) => p) /\ (at(S2) => ¬p))

after(S) <=> [after(S1) V after(S2)]

at(S) /\ p => <>at(S1)

at(S) /\ ¬p => <>at(S2)

S-4: If S is

BEGIN S1; S2;...;Sm END

where each Si is known to terminate, then

at(S) <=> at(S1)

after(S) <=> after(Sm).

Control cannot reach a statement that has already been executed.

∀i $1 \leq i \leq m$, after(Si) => ∀j $1 \leq j \leq i$, ¬<>at(Sj)

By S-1, S-2 and T-2,

∀i $1 \leq i \leq m$, at(Si) => ∀j $i \leq j \leq m$, <>after(Sj).

S-5: If S is the statement

WAIT UNTIL C

where C is a boolean condition, then

at(S) => at(S) UNTIL C /\

<>(at(S) /\ C) => <>(after(S) /\ C)

S-6: Given a set of guarded commands

```
SELECT
 WHEN g1 DO a1;
OR
 WHEN g2 DO a2;
OR
 ...
 WHEN gm DO am;
END
```

the following inferences can be made for all i,j, $i \neq j$.

G-1: Commands are executed in the order in which guards are recognized to be true.

[gi ONLYAFTER gj] => [at(ai) ONLYAFTER at(aj)]

G-2: The underlying scheduler of processes is such that once a guard is true, the associated command will be eventually executed.

gi => <>at(ai)

G-3: A command can be executed only after its guard becomes true.

at(ai) ONLYAFTER gi

3.3 Definition of predicates on the state of the system

The following functions and predicates on the state of masters and slaves are utilized for making precise statements about the working of Delta's distributed executive. (Henceforth, for the sake of brevity, we will refer to a transaction with timestamp t as transaction t.

Variables starting with m refer to messages, p to masters, e to entities and t to transactions.)

master(t)	The master in charge of transaction t.
slaves(t)	The slaves that execute the actions for transaction t.
timestamp(T)	The timestamp of transaction T.
entities(t,c)	The data entities that are accessed by consumer c for transaction t.

3.3.1 Predicates associated with the execution of transactions

start(t)	Actions for the transaction with timestamp t are started by some master.
req(t,c)	Slave c has received a request for executing actions for transaction t.
access(t,c,e)	Entity e is accessed by slave c for transaction t.
term(t,c)	Slave c completes the actions for transaction t.
wait(t,c)	Transaction t is made to wait by slave c.
committed(t,c)	Slave c has made transaction t's changes permanent, has unlocked the entities used by t and has destroyed the context of t.
rolledback(t,c)	Slave c has unlocked the entities used by t and has made t to wait.
aborted(t,c)	Slave c has unlocked entities used by t and has destroyed the context of t.
need-to-abort(t,c)	Slave c needs to abort all changes made by transaction t.

The following assumptions are made about the execution of actions.

1: Actions can terminate only after all entities have been accessed.

term(t,c) ONLYAFTER ∀e∈entities(t,c) access(t,c,e)

2: Terminated actions remain terminated unless an abortion or rollback takes place.

term(t,c) => [term(t,c) UNTIL (rolledback(t,c) V aborted(t,c))]

3: If a transaction has terminated, then it cannot be waiting.

term(t,c) => ¬wait(t,c)

3.3.2 Predicates associated with message-passing

send(m,p1,p2)	Process p1 sends message m to process p2.
sent(m,p1,p2)	Message m has been sent from process p1 to process p2.
receive(m,p1,p2)	Process p2 receives message m from process p1.
received(m,p1,p2)	Message m from process p1 has been received by process p2.

If S is the statement executing which p1 sends m to p2 then

in(S) <=> send(m,p1,p2)
after(S) => sent(m,p1,p2)
sent(m,p1,p2) ONLYAFTER after(S)

Similarly, if p2 executes statement R to receive m from p1 then

in(R) <=> receive(m,p1,p2)
after(R) => received(m,p1,p2)
received(m,p1,p2) ONLYAFTER after(R)

The following assumptions are made about the communication medium

1: A message will be eventually received.

send(m,p1,p2) => <>receive(m,p1,p2)

2: A message can be received only after it is sent.

receive(m,p1,p2) ONLYAFTER send(m,p1,p2)

3: Message exchanges are permanent.

sent(m,p1,p2) => []sent(m,p1,p2)
received(m,p1,p2) => []received(m,p1,p2)

Along with assumption 2 above, these imply,

received(m,p1,p2) => sent(m,p1,p2)

3.3.3 Predicates associated with locks

lock(t,c,e)	Slave c locks entity e for transaction t.
locked(t,c,e)	Entity e has been locked by slave c for transaction t.
unlock(t,c,e)	Slave c unlocks entity e for transaction t.

conflict(t1,t2,c) Transaction t1 needs an entity that is already locked by slave c for transaction t2.

A data entity is not locked for a transaction until it is locked by a slave for that transaction.

¬locked(t,c,e) => [¬locked(t,c,e) UNTIL lock(t,c,e)]

If c executes statement L to lock entity e for t then

in(L) <=> lock(t,c,e)
after(L) => locked(t,c,e)

A data entity remains locked for a transaction until a slave unlocks it.

locked(t,c,e) => [locked(t,c,e) UNTIL unlock(t,c,e)]

If c executes statement U to unlock entity e used by transaction t then

in(U) <=> unlock(t,c,e)
after(U) => ¬locked(t,c,e)

Transaction t1 conflicts with transaction t2 at slave c if and only if t1 needs an entity locked for t2.

conflict(t1,t2,c) <=> ∃e∈entities(t1,c) locked(t2,c,e).

3.4 Definition of well-formed, two-phase and legal transactions

A transaction t is said to be well-formed if for every slave c that is invoked by the master of that transaction, for every entity e accessed by c, the following are true.

WF-1: A data entity can be accessed only if it is locked.

(access(t,c,e) => locked(t,c,e))

WF-2: A data entity is unlocked only after it is locked.

(unlock(t,c,e) ONLYAFTER locked(t,c,e))

WF-3: Once a data entity is locked, it will be eventually unlocked.

(lock(t,c,e) => <>unlock(t,c,e))

We assume that no change occurs when a slave tries to lock an entity for a transaction when it is already locked for that transaction.

A concurrent execution of a set of transactions {t1,t2,.....,tn} is said to be legal if

LE-1: ∀i $1 \leq i \leq n$, ∀c∈slaves(t), ∀e∈entities(t,c),
(lock(ti,c,e) => ∀j, j≠i, ¬locked(tj,c,e))

that is, when a slave locks A data entity, there is no other transaction for which that entity is locked. Thus, for concurrent transactions to be legal, a transaction should be made to wait if it needs to lock A data entity that is locked by a different transaction. (Recall that data is partitioned and replicated and that a particular partition is associated with a specific slave.)

A transaction t is said to be two-phase if

TP-1: The unlock phase starts only after locks are placed on all accessed entities.

∀c∈slaves(t), ∀e∈entities(t,c),
unlock(t,c,e) ONLYAFTER
∀c1∈slaves(t) ∀e1∈entities(t,c1) lock(t,c1,e1)

TP-2: Once A data entity has been unlocked for a transaction, it cannot be locked again by that transaction.

∀c∈slaves(t), ∀e∈entities(t,c),
unlock(t,c,e) => ¬<>lock(t,c,e)

4. PROOF OF CORRECTNESS OF DELTA'S DISTRIBUTED EXECUTIVE

Proof of correctness of Delta's distributed executive is based on the description given in section two and the definitions in section three. As mentioned in section 3, our proof will consist of two parts.

a) Showing that Delta's distributed executive controls the execution of transactions in such a way that consistency of data is maintained at all times.

b) Showing that a transaction once initiated by an agent will eventully terminate.

4.1 Transactions in Delta maintain consistency of data

We will prove that concurrent transactions permitted by Delta's distributed executive are serializable. Since serializability of transactions is a sufficient condition for the consistency of data, transactions in Delta preserve the consistency of distributed data. To demonstrate serializability, we use the following theorem from [3].

Theorem: If each member of a set of transactions {t1,t2,...,tn} is well-formed and two-phase, then a concurrent execution of transactions t1,t2,....,tn which is legal is serializable.

Thus to show that the transactions allowed by Delta's distributed executive are serializable and hence preserve the consistency of distributed data, we prove the following:

1. Transactions in Delta are well-formed.
2. Concurrent execution of transactions in Delta is legal.

3. Transactions in Delta are two-phase.

To prove (1), (2) and (3) for the Delta system, we consider a typical transaction in Delta and examine how its actions are executed by a master. Transactions that complete without interruptions such as rollbacks and aborts are first examined. (Consistency with rollbacks and aborts is discussed at the end of this subsection.) In this case, for each transaction, the following activities take place: A master sends requests for actions to slaves; After all such requests have been sent, the master sends "prepare to commit" messages to the slaves; After completing the actions for the transaction, slaves respond with a "ready to commit" message; After receiving such messages from all slaves, the master instructs them to commit their changes.

In the following statements, t refers to a typical transaction, and p to the master of t. Lci (Lpi) is a program label in the code for slave c (master p). Note that at any one time a slave could be engaged in executing the actions for different transactions. Hence it is necessary to qualify each action of the slave with the transaction corresponding to that action. However, in what follows we will be concerned with one particular transaction. Hence we do not make this qualification explicit in the proofs. In developing the proof, the statement following "--" provides justification for the preceding statement in the proof.

4.1.1 Transactions in delta are well-formed

Proof of WF-1: A data entity can be accessed only if it is locked.

$\forall c \in slaves(t), \forall e \in entities(t,c),$

(access(t,c,e) => locked(t,c,e))

Follows directly from the fact that a slave locks data items (via statements Lc1 and Lc9) just prior to access (via statements Lc2 and Lc10).

Proof of WF-2: Every entity is unlocked only after it is locked.

¥c∈slaves(t), ¥e∈entities(t,c),
(unlock(t,c,e) ONLYAFTER locked(t,c,e))

A data entity required for a transaction is unlocked only after a slave receives a commit message from the master of that transaction (statement 1 below). A master sends commit messages only after all slaves have indicated their willingness to commit (statement 2). A slave indicates its willingness to commit only after it has completed all accesses for that transaction (statement 3). Data is accessed for a transaction only after it has been locked for that transaction (statement 4). Thus a data entity is unlocked only if after it is locked.

Let c ∈ slaves(t), e ∈ entities(t,c).

1. [unlock(t,c,e) => in(Lc17)
/\ in(Lc17) ONLYAFTER receive("commit t",p,c)
/\ receive("commit t",p,c) ONLYAFTER send("commit t",p,c)]
=>
unlock(t,c,e) ONLYAFTER send("commit t", p, c)

-- Behavior of slaves in the absence of aborts and rollbacks,
definition of message-passing predicates,
semantics of guarded commands,
assumption 2 about the communication medium,
T-6, T-7.

2. [send("commit t",p,c) => in(Lp5)
/\ at(Lp5) ONLYAFTER
¥c1 ∈slaves(t), sent("ready to commit t, c1, p)]
=>
send("commit t", p, c) ONLYAFTER sent("ready to commit t", c, p)

-- definition of message-passing predicates,
assumption 2 about the communication medium,
behavior of master, definition of "at" and "in",
T-6, T-7.

3. [sent("ready to commit t",c,p) ONLYAFTER after(Lc13)
/\ after(Lc13) ONLYAFTER term(t,c)
/\ term(t,c) ONLYAFTER access(t,c,e)]
=>
sent("ready to commit t", c, p) ONLYAFTER access(t,c,e)

-- definition of message-passing predicates,
behavior of slaves,
assumption 1 concerning the nature of actions, T-6.

4. [access(t,c,e) =>
=> [in(Lc2) V in(Lc10)]
=> locked(t,c,e)]

-- behavior of slaves.

5. unlock(t,c,e) ONLYAFTER locked(t,c,e).

-- T-6, T-7, statements 1 to 4.

6. WF-2 is the generalization of 5.

Proof of WF-3: Once a data entity is locked, it will be eventually unlocked.

$$\forall c \in slaves(t), \forall e \in entities(t,c), (lock(t,c,e) \Rightarrow \Diamond unlock(t,c,e))$$

Once data needed for a transaction are locked, in the absence of aborts and rollbacks, the actions for that transaction will terminate (statement 1). Eventually slaves receive "prepare to commit" messages from the master (statement 2). Since actions have terminated, slaves respond with "ready to commit" messages (statement 3) and hence the master receives "ready to commit" messages from all slaves (statement 4). Thus the master instructs all slaves to commit their changes. Locks are then removed (statement 5). Thus all locked data entites will be eventually unlocked.

Let c $\in$ slaves(t), e $\in$ entities(t,c).

For a transaction to be executed, some master should initiate it. Thus initially start(t) and hence at(Lp1) will be true.

1. at(Lp1)
=> at(Lp2)
=> <>send("prepare to commit t", p,c)

=> <>receive("prepare to commit t",p,c)
=> <>at(Lc12)
=> <>(at(Lc12) UNTIL term(t,c))

-- behavior of master and slaves in the absence of aborts,
definition of message-passing predicates,
definition of the WAIT-UNTIL construct,
assumption 1 about the communication medium.

2. lock(t,c,e)
=> [in(Lc1) V in(Lc9))]
=> <>[after(Lc2) V after(Lc10)]
=> <>term(t,c)
=> <>[]term(t,c)

-- behavior of slaves, definition of lock predicates,
assumption 2 about actions.

3. <>(at(Lc12) /\ term(t,c))
=> <>at(Lc13)
=> <>send("ready to commit t",c,p)
=> <>receive("ready to commit t",c,p)

-- T-3, statements 1 and 2,
definition of the WAIT-UNTIL construct,
assumption 1 about the commmmunication medium,
behavior of slaves,
definition of message-passing predicates.

4. ∀c1∈slaves(t), <>receive("ready to commit,c1,p)

-- generalization of 3.

5. at(Lp1)
=> <>at(Lp3)
=> <>at(Lp5)
=> <>send("commit t",p,c)
=> <>receive("commit t",p,c)
=> <>unlock(t,c,e)

-- statement 4,
behavior of master in the absence of aborts and rollbacks,
behavior of slaves,
assumption 1 about the commmmunication medium,
definition of message-passing predicates.

6. lock(t,c,e) => <>unlock(t,c,e)

-- statements 1 to 5.

7. WF-3 is generalization of 6.

4.1.2 Concurrent transactions in Delta are legal

Proof of LE-1: Data entities are locked for a transaction only if they are not locked for some other transaction.

$$\forall i\ 1 \le i \le n, \forall c \in slaves(t), \forall e \in entities(t_i,c),$$
$$(lock(ti,c,e) \Rightarrow \forall j, j \neq i, \neg locked(tj,c,e))$$

A slave locks A data entity for a transaction only when it does not conflict with transactions in progress. Legality of transactions directly follows from the definition of conflict.

Let $c \in slaves(t)$, $e \in entities(t,c)$.

1. $lock(t1,c,e)$
   $\Rightarrow \forall t2\ [in(Lc1) \vee in(Lc9)] \wedge \neg conflict(t1,t2,c)$
   $\Rightarrow \forall e1 \in entities(t1)\ \neg locked(t2,c,e1).$
   $\Rightarrow \neg locked(t2,c,e).$

   -- behavior of slaves, definition of conflict.

2. LE-1 is the generalization of 1.

4.1.3 Transactions in Delta are two-phase

Proof of TP-1: Unlock phase follows the lock phase.

$$\forall c \in slaves(t), \forall e \in entities(t,c),$$
$$(unlock(t,c,e)\ ONLYAFTER$$
$$\forall c1 \in slaves(t)\ \forall e1 \in entities(t,c1)\ lock(t,c1,e1))$$

Slaves make all changes permanent and unlock the locked entities onlyafter they receive the commit message from the master (statement 1). A master sends a commit message only if all slaves are ready to commit, (statement 2). A slave indicates its readiness to commit only if it has completed access (statement 3). Before accessing data, a slave locks the data entities (statement 4). Therefore, all locking is done prior to the beginning of the commit protocol whereas all unlocking is done at the end of the commit protocol.

Let $c \in slaves(t)$, $e \in entities(t,c)$.

1. [unlock(t,c,e) <=> in(Lc17)
   /\ in(Lc17) ONLYAFTER receive("commit t",p,c)]
   =>
   unlock(t,c,e) ONLYAFTER receive("commit t", p, c)

   -- behavior of slaves in the absence of aborts and rollbacks,
      definition of guarded commands, T-7.

2. send("commit t",p,c) ONLYAFTER
   ∀c1∈slaves(t), received("ready to commit t",c1,p)
   =>
   send("commit t",p,c) ONLYAFTER send("ready to commit t",c,p)

   -- behavior of master in the absence of aborts,
      assumption 3 about the communication medium, T-6.

3. [send("ready to commit t",c,p) <=> in(Lc13)
   /\ in(Lc13) ONLYAFTER term(t,c)
   /\ term(t,c) ONLYAFTER ∀e1∈entities(t,c), access(t,c,e1)]
   =>
   send("ready to commit t",c,p) ONLYAFTER access(t,c,e)

   -- definition of message-passing predicates,
      assumption 1 about actions,
      T-6, T-7, behavior of slaves.

4. access(t,c,e)
   => [in(Lc2) V in(Lc10))
   => ∀e∈entities(t,c),locked(t,c,e)

   -- behavior of slaves, definitions of lock predicates.

5. unlock(t,c,e) ONLYAFTER
   ∀c∈slaves(t), ∀e∈entities(t,c), locked(t,c,e)

   -- statements 1 to 4, T-6, T-7.

6.TP-1 is the generalization of 5.

Proof of TP-2: An unlocked data entity is never again locked for that transaction.

∀c∈slaves(t), ∀e∈entities(t,c),
unlock(t,c,e) => ¬<>lock(t,c,e)

A slave unlocks data locked for a transaction only after it receives a commit message from the master for that transaction (statement 1). Once a master has sent commit messages to the slaves, it cannot demand actions from them (statement 2). Entities are locked only after a

master demands actions from slaves (statement 3). Thus data will not be locked for a transaction once locks set for that transaction have been removed (statement 4).

Let c ∉ slaves(t), e ∉ entities(t,c).

1. unlock(t,c,e)
   => in(Lc17)
   => received("commit t",p,c)
   => sent("commit t",p,c)

   -- behavior of slaves in the absence of rollbacks and aborts, assumption 3 about the communication medium.

2. sent("commit t",p,c) ONLYAFTER after(Lp6)
   => sent("commit t",p,c) ONLYAFTER []¬at(Lp2)

   -- Definition of message-passing predicates, application of S-4 to master code.

3. unlock(t,c,e) ONLYAFTER []¬at(Lp2)

   -- statements 1 and 2, T-7.

4. [lock(t,c,e) ONLYAFTER receive("request actions for t",p,c)
   /\ send("request actions for t",p,c) ONLYAFTER at(Lp2)]
   =>
   lock(t,c,e) ONLYAFTER at(Lp2)
   =>
   ¬lock(t,c,e) UNTIL at(Lp2)

   -- Definition of message-passing predicates, behavior of slaves in the absence of aborts and rollbacks, T-6, assumption 2 about the communication medium, definition of ONLYAFTER.

5. unlock(t,c,e) => []¬at(Lp2) => ¬<>lock(t,c,e)

   -- T-5, statements 1 to 4.

6. TP-2 is the generalization of 5.

So far, we have considered the case where a transaction completes without intervening abortions or rollbacks. Let us briefly consider rollbacks and abortions now. Before a slave performs an action for a transaction it locks all data entities needed for that transaction. Any changes to the data are committed only after the slave receives a commit

message from the master. Since a rollback occurs prior to the receipt of a commit message and since a slave releases all locks set by a rolledback transaction, the state of the relevant data entities after rollback is exactly the same as that which existed before the locks were set. Hence consistency is not affected by a rolledback transaction. Similarly, an aborted transaction terminates without modifying any data. Thus actions performed for rolledback and aborted transactions are in effect null operations, thereby not affecting the consistency of data.

4.2 Transactions in Delta will terminate.

A transaction submitted by an agent is said to have terminated when all slaves involved in that transaction have committed their actions. Thus, to demonstrate termination, the following should be proved.

$\forall t, [](start(t) \Rightarrow \forall c \leftarrow slaves(t), <>committed(t,c))$.

Proof of termination is carried out in three steps. In step I of the proof, we show that in the absence of conflicts and failures, every transaction will eventually terminate. In step II we assume that conflicts are possible and examine how Delta's conflict resolution strategy performs with respect to termination of transactions. The conflict resolution strategy used in Delta prevents deadlocks by favoring that transaction with the smallest timestamp. This strategy also avoids situations where a transaction waits indefinitely since there are only a finite number of transactions that could have started before any waiting transaction. In step III we approach the problem of demonstrating termination in the presence of failures through successive

relaxation of the assumptions concerning the nature of transactions and the functioning of the system itself. As we shall see, the final assumptions, relaxing any one of which cannot guarantee termination, indicate those system components whose failure may affect termination.

4.2.1 STEP-I: Transactions will terminate if they do not conflict with one another.

By examining the code for masters and slaves, the following inferences can be made: Once a master has been invoked, slaves will eventually receive the requests for action (statements 1 and 2 below). In the absence of conflicts, no transaction is aborted, rolledback, or made to wait (statement 3). In the absence of waits all actions terminate successfully (statement 4). Thus in response to its "prepare to commit" message, a master receives "ready to commit" messages from all slaves at which point the master instructs them to commit the changes (statements 5 and 6).

Let c ∉ slaves(t), e ∉ entities(t,c).

1. start(t) <=> at(Lp1)

2. start(t)
   => <>after(Lp1)
   => <>sent("request action for t",p,c)
   => <>received("request action for t",p,c)
   => <>req(t,c)

   -- definition of message-passing predicates,
      behavior of master and slaves,
      assumption 1 about communication medium,
      definition of "req".

3. In the absence of conflicts,

   ∀t1, []¬conflict(t,t1,c)
   => []¬[at(Lc5) ∨ at(Lc6) ∨ at(Lc7)]
   => []¬[after(Lc5) ∨ after(Lc6) ∨ after(Lc7)]
   => []¬wait(t,c) /\ []¬rolledback(t,c) /\ []¬aborted(t,c)

-- T-2, behavior of slaves.

4. start(t) => <>req(t,c) => <>term(t,c) => <>[]term(t,c)

-- statements 2 and 3,
assumption 2 about actions,
T-4.

5. start(t)
=> <>after(Lp2)
=> ¥c1∈slaves(t), <>sent("prepare to commit t",p,c1)
=> <>received("prepare to commit t",p,c)
=> <>at(Lc11)
=> <>at(Lc12)

-- statement 3,
behavior of master and slaves,
assumptions about the communication medium,
semantics of guarded commands.

6. start(t)
=> <>[~~at(Lc12)~~ /\ term(t,c)]
=> <>at(Lc13)
=> <>sent("ready to commit t", c,p)
=> <>received("ready to commit t", c,p)
=> ¥c1∈slaves(t), <>received("ready to commit t", c1, p)
=> <>after(Lp3)
=> <>at(Lp5)
=> <>sent("commit t",p,c)
=> <>received("commit t",p,c)
=> ~~<>after(Lc18)~~
=> <>committed(t,c)

-- statements 4 and 5,
definition of message-passing predicates,
semantics of the WAIT-UNTIL construct,
assumptions about the communication medium,
semantics of guarded commands,
behavior of master and slaves.

7. generalization of statement 6 for all slaves proves termination in the absence of conflicts and failures.

4.2.2 STEP-II: Transactions will terminate even if conflicts occur.

As seen earlier, in Delta, conflict between two transactions is resolved using the following strategy: If the younger of the conflicting

Since the Proposal to commit message is sent via a tpsch thread/ message to all slaves, all slaves enter the commit protocol simultaneously.

* indicated their readiness to commit

(i) older $t_{m'}$; or

(ii) both.

In case (i) the t_m in L will eventually commit (once the master releases all locks held by $t_{m'}$ on the conflicting objects) and t_m in L (i.e. we are assuming without loss of generality that (b) are assumed to commit in the hypothesis

$\#\#$ $\forall$ $\exists t_{k'} > t_2$ conflict $(t_2, t_{k'}, c) \wedge$ ready-to-commit $(t_{k'}, c)$

transactions is in progress, it is either rolled back or aborted. If the older transaction is in progress, the younger transaction is made to wait. First we consider termination when rollback is the adopted strategy. Abortion is considered later.

Proof will be by induction on t. This necessitates showing the following.

(a) start(1) => ∀c∈slaves(1), <>committed(1,c)

(b) If it is assumed that for some t2
∀t1, t1<t2, ∀c1∈slaves(t1), [start(t1) => <>committed(t1,c1)]
then
∀c2∈slaves(t2), [start(t2) => <>committed(t2,c2)].

Proof of (a)

Since transaction 1 is the oldest transaction in the system, even if it conflicts with other transactions it is never made to wait. Hence, by the proof in step I, transaction 1 will terminate.

Proof of (b)

Assume the hypothesis of induction step. Transaction t2 may or may not conflict with older transactions. If it does not, then by the result of step I, t2 will terminate. Otherwise, since transactions with lower timestamps are assumed to terminate, eventually t2 no longer conflicts with older transactions (statements 1 and 2). t2 becomes the oldest waiting transaction at which point its actions will be performed (statement 3). Future conflicts, if any, will occur only with transactions younger than t2 and by the conflict resolution strategy t2 will not be made to wait (statement 4).

Assume t2 has a conflict with an older transaction t3.

1. ∃c∈slaves(t2), ∃t3<t2, conflict(t2,t3,c).
=>

<>wait(t2,c)

-- behavior of slaves.

2. ∀c∈slaves(t3), <>committed(t3,c)
=> ∀e∈entities(t,c), <>¬locked(t3,c,e)
=> <>¬conflict(t2,t3,c)

-- behavior of slaves,
assumption in the induction step,
definition of conflict.

3. ∀t3<t2, ∀c∈slaves(t3), <>committed(t3,c)
=> ∀t3<t2, ∀c, []¬wait(t3,c)
=> <>¬wait(t2,c)

-- statement 2, behavior of slave c when t2 becomes
the oldest waiting transaction.

4. ∀t1<t2, [¬wait(t2,c) /\ committed(t1,c)]
=>
[]¬wait(t2,c)

-- t2 is the oldest extant transaction.

5. ∀c∈slaves(t), <>committed(t2,c).

-- using the result of step I.

So far, we assumed rollback as the conflict resolution strategy and showed that in spite of conflicts a transaction will terminate. Now we show that even if abortion of transactions is resorted to to resolve conflicts, transaction will terminate. An aborted transaction is resubmitted with the same timestamp. The resubmitted transaction is executed as though it were submitted for the first time. As in the proof of (b) above, eventually all older transactions commit, whence a previously aborted transaction will never conflict with older transactions. Thus by the proof in step I, this transaction will terminate.

It should be obvious that the overheads due to aborting a transaction

are more than that due to rollback. This is because abortion affects all the slaves whereas rollback is confined to the slave where the rollback occurs. The current implementation of Delta hence uses rollbacks for conflict resolution.

4.2.3 STEP-III: Termination occurs in spite of failures.

In a system such as Delta, slaves, masters or the communication medium may fail. We assume that the communication medium is robust and hence examine the termination of transactions in the event of failure of slaves and masters only.

Failure of slaves: Assume first that masters do not fail, but slaves may. If a slave fails, transactions which use the data controlled by that slave are aborted and resubmitted with their original timestamp. The transactions use only those slaves that are available. Given that information in Delta is partioned and replicated, by the proof of Step II, the resubmitted transactions will be able to terminate using the other copies of the data controlled by the failed slave. However, recovery may not be possible if all slaves that possess a copy of a partition fail simultaneously. When a failed node is eventually restarted, its data is restored to be consistent with the redundant copies of the partition that it controls.

Failure of a master: A master may fail before prepare to commit messages have been sent to all slaves involved in a transaction, or after. To facilitate recovery in the event of failure of a master, the list of slaves involved in a transaction is passed to the slaves along

with the prepare to commit message.

Case-1: Master fails before all prepare to commit messages have been sent. In Delta, a master's death is recognized by other masters through the virtual ring. Suppose p1 fails while controlling transaction t. Then some master p2 takes over, aborts t and resubmits it with the same timestamp. If the new master does not fail, then according to the results of steps I and II, termination is guaranteed. If p2 fails, as in the case of p1, another master becomes the new master for t. As long as there is some master which assumes control of t and does not fail before t is committed, then based on the discussion of failure of slaves, termination is guaranteed.

Case-2: A master fails after prepare to commit messages have been sent to all slaves. Slaves communicate to decide on the course of action.

a. If at least one slave has committed, then all commit.
b. If all have indicated their readiness to commit, then all commit.
c. If at least one slave has aborted, then all abort.
d. If none of the above situations apply, then the transaction is aborted and resubmitted by another master.

If some slaves have failed and if the slaves that are up have neither committed nor aborted, then they wait for the failed slaves to restart and then decide based on the above criteria. Earlier, we showed that even aborted transactions terminate. Hence even in situations (c) and (d) above, a transaction will eventually terminate. Hereagain, termination is guaranteed as long as there is some master which assumes control of t and does not fail before t is committed.

5. DISCUSSION

We have demonstrated the correctness of the distributed concurrency control mechanism of Delta. In order to accomplish this task, the following were needed:

a. Definition of the working of Delta's distributed executive. While encompassing the essential details, this definition had to be precise and abstract enough to permit formal reasoning.

b. Criteria for the correctness of the distributed executive.

By examining the description of the distributed executive in [5], we came up with the abstract programs for masters and slaves given in section 2. Notice that code for masters is essentially a sequential program while the code for slaves is a set of guarded commands. The assumptions made by LeLann were precisely stated through temporal logic statements.

The criteria established for the correctness of Delta's distributed executive were, (1) concurrent transactions should maintain the consistency of distributed data and (2) every transaction should eventually terminate.

The notion of serializability was utilized to demonstrate that transactions in Delta satisfy the consistency requirement. By proving that transactions in Delta are well-formed, two-phase and legal, it was shown that transactions in Delta are serializable. Since serializability is a sufficient condition for consistency [3], Delta's

distributed executive satisfies the consistency criterion.

Commitment of all changes made by a transaction was taken to be equivalent to the termination of the transaction. As is well known, termination cannot always be guaranteed in an environment where processing or storage elements may fail. So we approached the proof of termination with the purpose of identifying those situations for which termination can not be proven. This necessitated a three-step approach. In the first step, transactions were assumed not to conflict, in the second, conflicts were permitted but component failures were not, and in the last, component failures were taken into consideration. Proof of termination was undertaken through systematic examination of the flow of control through masters and slaves. Consideration of termination showed that:

a. Even if slaves fail, utilizing the redundancy in the system a failed slave's state can be restored to be consistent with the rest of the system. Thus slaves' failure does not affect termination unless all slaves that hold a copy of a partition fail simultaneously.

b. Failure of masters is liable to affect termination (for example, if every master that takes control of a transaction fails before sending the prepare to commit message to all slaves).

Thus, except in pathological cases, transactions will terminate even if masters and slaves fail, or conflicts exist.

Our attempt to verify an abstract view of Delta, points out to the use of verification even at the design stage for systems, i.e., even before the system has been implemented. For this to be possible, the assumptions underlying the system should be stated precisely, and an abstract yet precise description of the system should be available.

Delta has been designed to serve in a real-time environment. Our proof of termination only demonstrates that every transaction will eventually commit its results and thus terminate. However, typically for real-time systems, a bound is placed on acceptable delays before a transaction is completed. Whether Delta does satisfy such requirements can only be determined through simulation.

Temporal logic served as the tool to formalize our assumptions and provide rigor to the proof. The fact that using the same formalism we were able to demonstrate both consistency and termination properties of Delta attests to the usefulness of temporal logic in dealing with invariant and liveness properties of software systems.

ACKNOWLEDGEMENTS

Our thanks to Le Lann for his explanation of Delta's distribute executive, and to David Stemple, Alexander Wolf and Laurie Dillon for their comments on an earlier version of this paper.

7. REFERENCES

[1] Ben-Ari, M. and Pnueli, A., "Temporal Logic proofs of Concurrent Programs", Technical report 80-44, Tel Aviv University, Jan 1981.

[2] Bernstein, P.A. and Shipman, D.W., "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases", ACM Transactions on Database Systems 5, 1, March 1980, 52-68.

[3] Eswaran, K.P. et al., "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM 11, Nov 1976, 624-633.

[4] Lamport, L., "'Sometime' is Sometimes 'Not Never'", Proc. Seventh Annual Symposium on POPL, Jan 1980, 174-185.

[5] Le Lann, G., "A Distributed System for Real-Time Transaction Processing", IEEE Computer, Feb 1981, 43-48.

[6] Owicki, S. and Lamport, L. "Proving Liveness Properties of Concurrent Programs", ACM Transactions on Programming Languages and Systems, 4, (July 1982) 455-495.

[7] Pnueli, A., "The Temporal Semantics of Concurrent Programs", in "Semantics of Concurrent Computation", Springer Lecture Notes in Computer Science 70, June 1979, Springer-Verlag, 1-20.

[8] P.M. Rosenkrantz, et al., "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database Systems 2, June 1978, 178-198.

[9] Rothnie et al. Introduction to a System for Distributed Databases (SDD-1). ACM TODS 5,1, (March 1980), 1-17.

[10] Schwartz, R.L. and Melliar-Smith, P.M., "Temporal Logic Specifications of Distributed Systems", Proc. of the Second International Conference on Distributed Systems, Apr 1981.

[11] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", IEEE Transactions on Software Engineering, SE-5,3, May 1979, 188-194.

[12], R.H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Transactions on Database Systems 2, June 1979, 180-209.

adapted for Imbert earlier