RJ2895 (36591) 8/7/80
Computer Science

# Research Report

## A TRANSACTION MODEL

Jim Gray

IBM Research Laboratory
San Jose, California 95193



IBM Research Division
Yorktown Heights, New York · San Jose, California · Zurich, Switzerland

Copies may be requested from:
IBM Thomas J. Watson Research Center
Distribution Services
Post Office Box 218
Yorktown Heights, New York 10598

RJ2895 (36591) 8/7/80
Computer Science

# A TRANSACTION MODEL

Jim Gray

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: This paper is an attempt to teresly restate several theoretical results about transaction recovery and concurrency control. A formal model of entities, actions, transactions, entity failures, concurrency and distributed system is required to present these results. Included are theorems on transaction undo and redo, degrees of consistency, predicate locks, granularity of locks, deadlock, and two-phase commit.

# A TRANSACTION MODEL

Jim Gray

IBM Research San Jose Research Laboratory
San Jose, California. 95193
February 1980

## ABSTRACT:
This paper is an attempt to tersely restate several theoretical results about transaction recovery and concurrency control. A formal model of entities, actions, transactions, entity failures, concurrency and distributed system is required to present these results. Included are theorems on transaction undo and redo, degrees of consistency, predicate locks, granularity of locks, deadlock, and two-phase commit.

## CONTENTS



## ACKNOWLEDGMENTS:
This paper draws heavily from the models of Charlie Davies, Janusz Gorski, Butler Lampson and Howard Sturgis, and from discussions with Dieter Gawlick, Bruce Lindsay, Ron Obermarck, and Irv Traiger. Critical readings by Bruce Lindsay and Anita Jones clarified several aspects of the presentation.

## DEFINITION OF TRANSACTION

A **database state** is a function from names to values. Each <name,value> pair is called an **entity.** The system provides **operations** each of which manipulates one or more entities. The execution of an operation on an entity is called an **action.** Record and terminal are typical entity types and read and write are typical operations.

Associated with a database is a predicate on entities called the **consistency constraint.** A database state satisfying the consistency constraint is said to be **consistent.**

Transactions are the mechanism which query and transform the database state. A **program** P is a static description of a transaction. The consistency constraint of the database is the minimal precondition and invariant of the program. The program may have a desired effect which is expressed as an additional postcondition C'. Using Hoare's notation:

$$C\ P\}\ C\ \&\ C'.$$

The execution of such a program on a database state is called **transaction** on the state. The exact execution sequence of a program is a function of the database state but we model a transaction as a fixed sequence of actions:

$$T = \langle\langle t,Ai,Ni\rangle \mid i=1,\ldots n\rangle$$

where t is the transaction name, Ai are operations and Ni are entity names.

The system may interleave the execution of the actions of several transactions. The execution of a set of transactions by the system starting from some database state is called a **history** and is denoted by the sequence:

$$H = \langle\langle ti,Ai,Ni\rangle \mid i=1,\ldots,m\rangle \quad .$$

which is an order preserving merge of the actions of the transactions. (A later section will show that even multiple nodes executing actions may be modeled by a single execution sequence.)

The users of the system author programs and invoke them as transactions. They are assured that each invocation:

- Will be executed exactly once (reliability).
- Will be isolated from temporary violations of the consistency constraint introduced by actions of concurrently executing transactions (consistency).

The transaction may attempt to commit with the consistency constraint violated or the program itself may detect an error. In this case the effects of the transaction are undone and the system or program issues an error message as the transaction output.

This paper presents a model of reliability and concurrency issues associated with such systems.

## RELIABILITY

### Model of failures

Reliability is a goal which may only be approached by the careful use of redundancy. One never has a reliable system, if everything fails then there is no hope of reconstructing which transactions executed or the final system state. Hence one needs a model of failures in order to discuss reliability.

There are three kinds of entities:

- **Real** entities initially have null values and their values cannot be changed once they are non-null. They may spontaneously change (in which case they are **real input** messages). Or they may be written once (in which case they are **real output** messages). If a transaction gives away a 100$ bill, that piece of paper exists and is beyond the control of the system.

- **Stable** entities have values which may be changed by the system and which survive system restart. Pages of duplexed disk or tape (two independent copies) are examples of stable storage. Pages of disk with an associated duplexed log of changes from a stable (archive) base version of the pages is another example of stable storage.

- **Volatile** entities have values which may be changed by the system and which are reset to null at system restart.

Two kinds of failures are considered:

- **Transaction restart:** for some reason a transaction needs to be restarted; however, its current state and the current state of the system exists (deadlock is an example of such a failure).

- **System restart:** for some reason the state of all volatile entities spontaneously change to null. However, all actions on stable and real entities prior to a certain instant will complete and all actions after that instant will have no effect until the system restarts.

The third kind of failure in which stable entities spontaneously change is not considered.

### Transaction restart

A transaction may experience a finite number of transaction restarts. The system must have some way to undo such partially executed transactions. It would be nice to postulate:

- Every action <t,A,E> has an undo-action ¬<t,A,E> which cancels the effect of the action.

Thus if T =<<t,Ai,Ei>|i=1,...,n> executes actions <<t,Ai,Ei>|i=1,...,k> for some k<n then the system executes the actions: <¬<t,Ai,Ei>|i=k,...,1> and the transaction is undone.

Unfortunately, some entities are real and actions on them cannot be undone. The action <John,Eat,Cake> has no undo-action ¬<John,Eat,Cake>. You can't have your cake and eat it too.

Hence transactions are partitioned into two parts delimited by a **commit action, c:**

$$T = \langle\langle t,Ai,Ei\rangle \mid i=1,\ldots,c-1\rangle \langle\langle t,Ai,Ei\rangle \mid i=c,\ldots,n\rangle$$

where <t,Ac,Ec> is the first action of T which has no undo-action. The first part of the transaction is called the **prelude** and the second is called the **commitment.**

The commitment is not undoable (precludes transaction undo). So the actual execution of a transaction will be of the form:

$$P1\ \neg P1\ P2\ \neg P2\ \ldots\ Pj\ \neg Pj\ T$$

where each Pi is a prefix of the prelude of T, and ¬Pi is the corresponding undo of that prefix. The transaction finally runs to completion without any more restarts. This backtracking of the transaction is transparent to the programmer who wrote T and will only be visible to the user of T if some Pi sends messages to the user which are subsequently undone.

In order to support transaction restart [3,4,5,6],

- Before executing any action on an entity, the system records the corresponding undo-action in an entity called the transaction UNDO log.

- When a transaction is restarted, the actions in the transaction's UNDO log are applied (in last-in first-out order) to reverse the effects of the transaction.

A program may fail at any time or may attempt to commit with the consistency constraint violated. In this case the system must undo the transaction and commit it with an error message. Thus the system cannot allow a transaction to enter the commitment phase until the the program issues its last action (recall that commitment cannot be undone). Hence a special operation, the **commit operation,** is introduced which signals the end of the program.

Prior to the commit action, the effects of all actions which modify real entities are **deferred** and placed into an entity called a **REDO log** or **intentions list** for the transaction. When the transaction issues an action <t,A,e> on real entity e, a **redo action** denoted #<t,A,e> is inserted into the REDO log and the action itself is deferred (thereby making the action undoable). The commit action of the transaction requests the system to make a stable copy of the REDO log and then perform all the deferred actions in the REDO log.

Some systems defer all actions until commit and thereby avoid the need for transaction undo. A formal definition of this is complex since one must define the values of volatile and stable entities which have deferred actions pending against them. (i.e. are the changes visible before commit?) Real entities are initially null and, as will be explained below, changes to them are deferred until they are committed.

## System restart

If there were no volatile entities, system restart would be of no consequence. However, current and anticipated hardware and software, provide microsecond access times to volatile storage and millisecond access times to stable storage. This gap is likely to persist. Hence the state of the transaction and of many entities are represented in volatile storage.

There are four kinds of transactions at system restart:

1. Those which have all their committed actions reflected in real and stable entities.

2. Those which were in the commitment phase at the instant of system restart.

3. Those which were in the prelude at the instant of the system restart.

4. Those which had not yet begun.

System restart transforms transactions of type 3 to transactions of type 4 because the (volatile) transaction state has been lost. System restart transforms transactions of type 2 to transactions of type 1 because committing transactions cannot be undone.

When the restart process completes, all transactions will be either committed or not-yet-started and the database state will satisfy the consistency constraint. The processing of the not-yet-started transactions is then begun. (The next section discusses reconstruction of volatile entities at restart.)

The previous section described two things necessary for transaction restart:

- When performing an action on a volatile or stable entity, record the undo action in the transaction's undo log.

- Defer all actions which have no undo action into a REDO log of actions until a transaction commits.

In order to accomplish system restart three more things must be done during normal operation of the system (in addition to the logging required for transaction restart) [5,6,8]:

- Before performing an uncommitted action on a stable entity, record the undo of the action in stable storage (in a stable UNDO log).

- As the first step of the commit of a transaction, record the (committed) REDO log of a transaction as a stable object.

- As the second step of the commit of a transaction, perform all the deferred actions in the REDO log and mark them done in the REDO log.

The first rule (called the **write ahead log** protocol) allows any uncommitted action on stable storage to be undone by applying the undo actions in the UNDO log. There is one problem: recording the undo record and performing the operation on the stable entity are two separate actions. If the system restarts

after the undo record is recorded but before the uncommitted operation is performed on the stable entity then the undo action will be applied to the old value (not to the new value) of the stable entity. Further, if a system restart occurs during transaction undo then the transaction may be only partially undone. To solve these problems,

- all undo actions ¬<t,A,E> must be **restartable** (or **idempotent**):
  <t,A,E>¬<t,A,E> is equivalent to <t,A,E>¬<t,A,E>,...,¬<t,A,E>
  is equivalent to ¬<t,A,E>,...,¬<t,A,E>

(i.e. DO-UNDO equals UNDO equals DO-UNDO-UNDO-...-UNDO). Restartability is an additional requirement if UNDO is to be part of restart. One way to avoid this problem is to defer all stable updates to the commitment phase and thereby eliminate UNDO at restart.

Deferring all actions which have no undo action allows the transaction to be undone at any time prior to that step (maximally defers commitment). The requirement that the REDO log be made a stable object at the commit step of the transaction allows the system to continue the execution of transaction commit at system restart (by simply applying the REDO log of the transaction). Once a deferred action is actually performed, the redo action may be marked as "done" in the REDO log. But again there is the problem that performing the operation and marking it "done" in the REDO log are two separate actions. At restart one cannot be sure whether the last deferred action is done or not. Hence, one concludes that:

- redo actions #<t,A,E> must be **restartable**:
  <t,A,E> is equivalent to <t,A,E>,#<t,A,E>,...,#<t,A,E>
  is equivalent to #<t,A,E>,...,#<t,A,E>

(i.e. DO equals REDO equals DO-REDO-REDO-...-REDO). Thus if restart redoes already-done actions, they will do no harm.

The most common technique for achieving restartability of redo and undo actions is to put a version number in the entity and in the undo or redo action. The redo (or undo) step tests the version number of the entity against the desired version number and does nothing if the version numbers match, otherwise the undo or redo step is actually performed. For example, message sequence numbers are used in this way to detect and discard duplicate (redo) messages and to cancel messages which have been sent (undo).

## Checkpoint and volatile entity reconstruction

In the discussion above, only the state of stable and real entities is reconstructed at system restart. The state of volatile entities is lost at system restart.

If the state of some volatile entity is to be reconstructed at system restart then the system must keep a REDO log of all actions on such entities and either:

- redo all actions on the entity since the beginning of time (using the REDO logs), or

- record a stable copy of the volatile entity at some time and then (using the REDO logs) redo all actions since that time.

Such a stable copy is called a checkpoint and is used to minimize redo work at restart [5,6].

The system allows the declaration of **recoverable volatile** entities. All entities appearing in the system invariant should be recoverable. The system periodically checkpoints recoverable volatile entities to stable objects. At restart all actions subsequent to the checkpoint on these entities are redone (including undo-actions).

Recoverable volatile entities are stable entities which have the fast access times of volatile entities. Their cost is periodic checkpoints, long term maintenance of REDO logs, and extra work at system restart.

## CONCURRENCY

### Motivation for serializable history

The initial database state satisfies the consistency constraint. Although each transaction preserves the consistency constraint, the constraint may be violated while the transaction is in progress. For example, if a transaction transfers funds from one account to another, the constraint that "money is preserved" may be violated between the debit of one account and the credit of another account.

If there is no concurrency then each transaction begins with a consistent state and produces a consistent state. However, one transaction may see inconsistencies introduced by another if transactions execute concurrently.

It is difficult to write programs which work correctly in the presence of such inconsistencies. Therefore the system prevents such inconsistencies. Clearly a history without any concurrency (e.g. the history T1•T2•...•Tn) has no concurrency anomalies. Such histories are called **serial histories.**

Concurrency is allowed, only if it does not introduce inconsistencies. The simplest definition of this is to insist that any allowable history be equivalent to a serial history. Several different equivalence relations have been defined. Perhaps the most intuitive is developed as follows: Two operations on entities are recognized:

- READ: reads the value of a named entity but does not change it.

- WRITE: writes the value of a named entity.

Given this interpretation, we define the **dependency relation of history:**

$$H = \langle \ldots, \langle t1, A1, e \rangle, \ldots, \langle t2, A2, e \rangle, \ldots \rangle$$

where $t1 \neq t2$ as:

$$DEP(H) = \{ \langle t1, e, t2 \rangle \mid (A1 = \text{WRITE and } A2 = \text{WRITE}) \text{ or } (A1 = \text{WRITE and } A2 = \text{READ}) \text{ or } (A1 = \text{READ and } A2 = \text{WRITE}) \}$$

DEP(H) tells "who gave what to whom". Two histories are **equivalent** if they have the same dependency relation. A history equivalent to a serial history is variously called **serializable, consistent,** and **degree 3 consistent.**

Intuitively, if H has the same "who gave what to whom" relationship as some serial history, then transactions cannot distinguish H from that serial history.

## Locking protocol for serializable histories

Locking is one technique for controlling concurrency (the interleaving of actions of several transactions). Used properly it can assure that all histories are equivalent to a serial history. Four new operations are introduced:

- LOCK_S: lock the named entity in shared mode.
- LOCK_X: lock the named entity in exclusive mode.
- UNLOCK_S: unlock the named entity from shared mode.
- UNLOCK_X: unlock the named entity from exclusive mode.

We say a lock action (S or X) by a transaction on the entity named E **covers** all actions up to the next unlock action (S or X respectively) action by that transaction on entity E.

The system will ensure that no LOCK_X action is performed on an entity while another transaction has that entity locked and conversely that no LOCK_S action on an entity is performed while another transaction has that entity locked in exclusive mode. More formally, the history H is **legal**

if H = <...<t1,A1,e>...<t2,A2,e>....> and t1 ≠ t2
and <t1,A1,e> is a lock action covering action <t2,A2,e> then:
if A1 = LOCK_S implies A2 ≠ LOCK_X
if A1 = LOCK_X implies (A2 ≠ LOCK_S and A2 ≠ LOCK_X).

A transaction is said to be **well-formed** if

- Each READ action is covered by a LOCK_S or LOCK_X action on the entity name to be read, and
- Each WRITE action is covered by a LOCK_X action on the entity to be written, and
- Nothing is covered beyond the last action of the transaction (i.e. it unlocks everything).

A transaction is said to be **two-phase** if it does not perform a lock action after the first unlock action.

The definition of DEP(H) and of equivalence given before must be amended to treat LOCK_S and UNLOCK_S actions as READ actions and LOCK_X and UNLOCK_X actions as WRITE actions. Given that amendment, the central theorem of this development is:

THEOREM[1] [2,10,11,12]:
(1) If all transactions are two-phase and well-formed
then any legal history is equivalent to a serial history.
(2) If some nontrivial§ transaction T is not two-phase or well-formed
then there is a transaction T' such that
T,T' have a legal history not equivalent to any serial history.

By automatically inserting LOCK_S and LOCK_X actions into a transaction prior to each READ and WRITE the system can guarantee a consistent execution of the transactions. Further, if the set of transactions is not known in advance all these precautions are required. However, if the set of transactions is known in advance, then some of the locks may be superfluous. For example, if there is only one transaction in the system then no locks are required. These observations have led to many variations of the theorem. Another source of variations is possible by giving the operations an interpretation (e.g. we interpreted read, write and lock).

## Locking and recovery

Consistency requires that a transaction be two-phase. We now argue that support of transaction restart requires that the second locking phase be deferred to transaction commit.

The first argument is based on the observation that UNLOCK_X generally does not have an undo action (and hence must be deferred). If transaction T1 unlocks an entity E which T1 has modified, entity E may be subsequently read or modified by another transaction T2. Restarting transaction T1 requires that the action of T1 on E be undone. This may invalidate the read or write of T2. One might suggest undoing T2, but T2 may have committed and hence cannot be undone. This argues that UNLOCK_X actions are not undoable and must be deferred.

A second argument observes that both UNLOCK_S and UNLOCK_X actions must be deferred to the commit action if the system automatically acquires locks for transactions. Suppose the system released a lock held by transaction T on entity E prior to the commit of T. Subsequent actions by T may require new locks. The acquisition of such locks after an unlock violates the two-phase lock protocol.

Summarizing:

- Consistency combined with transaction restart requires that UNLOCK_X actions be deferred until the transaction executes the commit action.

- Consistency combined with automatic locking requires that all locks be held until the transaction executes the commit action.

---

[1] Excluded are the null transaction, transactions which consist of a single read action and associated locks, and transactions which have locks which do not cover any action.

## Degrees of consistency

Most systems do not provide consistency. They fail to set some required locks or release them prior to the commit point. The resulting anomalies can often be described by the the notions of degrees of consistency [4]. (A more appropriate term would be degrees of inconsistency.) A Degree 3 consistency was defined before as a protocol which acquires locks to cover all actions and holds all locks to transaction commit. It was shown to prevent concurrency anomalies.

In order to support transaction restart, all systems acquire X-mode locks to cover writes and hold them to transaction commit. This is called the degree 1 consistency lock protocol.

If the system additionally acquires S-mode locks to cover reads but releases the locks prior to commit then it provides degree 2 consistency.

Both of these protocols are popular. Initially this was because system implementors did not understand the issues. Now some argue that the "lower" consistency degrees are more efficient than the degree 3 lock protocol. In the experiments we have done, degree 3 consistency has a cost (throughput and processor overhead) indistinguishable from the lower degrees of consistency.

## Predicate locks

Some transactions want to access many entities. Others want only a few. It is convenient to be able to issue one lock specifying a set of desired entities. Such locks are called **predicate locks** and are represented as <T,P,M> where T is the name of the requesting transaction, P is the predicate on entities, and M is a mode: either S (for shared) or X (for exclusive) [2]. A typical predicate is:

VARIETY = CABERNET and VINTNER = FREEMARKABBY and YEAR = 1971

This should reserve all entities satisfying this predicate. Two predicate locks <T1,P1,M1> and <T2,P2,M2> **conflict** (and hence cannot be granted concurrently) if:

- They are requested by different transactions (T1≠T2) and,
- The predicates are mutually satisfiable (P1&P2) and,
- The modes are incompatible (not both S-mode).

Predicate locks are an elegant idea. (People have tried to patent them!). Unfortunately, no one has proposed a acceptable implementation for them. (Predicate satisfiability was one of the first problems to be proven NP complete).

Another problem with predicate locks is that satisfiability is too weak a criterion for conflict. For example the predicate:

VARIETY = CABERNET and SEX = FEMALE

is only formally satisfiable (I think). But the predicate locks:

<T1,VARIETY=CABERNET,X> and <T2,SEX=FEMALE,X>

formally conflict. A theorem prover might sort this out, but theorem provers are suspected to be very expensive.

## Granularity of locking

The granularity of locks scheme captures the intent of predicate locks and avoids their high cost. It does this by choosing a fixed set of predicates.

Let P be a set of predicates on entities including the predicate TRUE and all predicates of the form: "ENTITYNAME=e" for each entity <e,v>. Assume that for each pair Q, Q' of predicates in P:

If for some entity e: Q(e) and Q'(e) are true (*)
then Q contains Q' or Q' contains Q

Define the binary relation → on P:

$$Q \rightarrow Q' \text{ iff for all entities e: } Q'(e) \text{ implies } Q(e).$$

The relation → is the set containment relation and because of assumption (*) above it orders P into a tree with root predicate TRUE.

Let the graph G(P) = <P,E> be the Hesse diagram of this partial order. That is P is the set of vertices and E is the set of edges such that:

$$E = \{ \langle A,B \rangle \mid A \rightarrow B \text{ and there is not C in P: } A \rightarrow C \rightarrow B \}$$

A new lock mode is introduced: **Intention mode (I-mode)** which is compatible with I-mode but not with S-mode or X-mode. Using this new mode, the following lock protocol allows transactions to lock any predicate Q in P:

- Before locking Q in S-mode or X-mode, acquire I-mode locks on all parents of Q on graph G(P).

If this protocol is followed, acquiring an S-mode or X-mode lock on a node Q **implicitly** acquires an S-mode or or X-mode lock on all entities e such that Q(e) is true.

THEOREM [4]: Suppose locks granted on graph G(P) are:

$$L = \{ \langle T,Q,M \rangle \}.$$

Define the **intent** of these locks to be:

$$L' = \{ \langle T,Q',M \rangle \mid \langle T,Q,M \rangle \text{ is in L and Q implies Q' and M} \neq \text{I-mod.} \}.$$

Then if no locks in L conflict, no locks in L' will conflict.

Since L' contains all the entity locks which are children of the predicate locks this indicates that the predicate locks prevent undesired concurrency. Here we have restricted the graph to a tree (by constraint (*) on P). These results generalize to an arbitrary set of predicates which in turn generate an arbitrary directed acyclic graph G. The generalization is useful but notationally complex. A more detailed development would also resolve I-mode into three modes IS, IX, and SIX for greater concurrency. see [4] for a development of these generalizations.

## Deadlock

A locking system must have some strategy for treating lock requests which conflict with already-granted locks. The simplest schemes either restart transactions which make such requests (no wait) or restart them if the request is ungranted for a certain time period (timeout).

Both of these approaches are subject to a phenomenon known as **livelock** in which two or more transactions repeatedly cause each other to be restarted. On the other hand, if transactions are allowed to wait indefinitely then they are subject to **deadlock** in which each member of a set of transactions is waiting for another member of the group to release a lock.

An approach which avoids such waiting allocates all desired resources to the transaction when it starts. This avoidance scheme has notorious performance because the resources potentially needed by a transaction are frequently much greater than those actually used.

Data management systems generally allow deadlock to occur. Deadlock appears as a cycle in the who-waits-for-whom graph. Deadlocks are resolved by choosing a minimal cost node set which breaks all cycles. Transactions corresponding to nodes of the set are undone and their locks preempted. The choice of preempted transactions must avoid livelock.

Fortunately, the mechanism for transaction restart is already present and so deadlock is simply another source of transaction restart. Further, deadlock is quite rare in practice (e.g. one transaction in one thousand) and the deadlock detection and resolution is comparatively simple (three pages of code compared to thirty for transaction restart).

We have observed that the probability a transaction deadlocks rises linearly with concurrency. A crude argument for this goes as follows: Let there be N+1 transactions each of which request r resources from a universe of R ($r \ll R$). The expected fraction of resources locked by others is $(Nr)/(2R)$ because each transaction holds about r/2 resources. Since a transaction makes r requests, the probability that it ever waits for a lock is: $(Nr^2)/(2R)$. Thus the probability that a request by a transaction will wait is proportional to N. Deadlocks are of the form "T waits for T' waits for T" or "T waits for T' waits for T'' waits for T", ... The probability of a cycle of length two involving T and T' is

$$P(T \text{ waits for } T')P(T' \text{ waits for } T).$$

which is:

$$(r^2/2R)(r^2/2R).$$

Since there are N possible T', the probability that T deadlocks with some T' in a cycle of length 2 is:

$$N(r^2/2R)^2.$$

Generalizing, the probablility of a cycle of any length is:

$$N(r^2/2R)^2 + N^2(r^2/2R)^3 + N^3(r^2/2R)^4 + \ldots$$

Assuming that the probability a transaction waits is much less than one (typically .1 to .001 in practice):

$$(Nr^2)/(2R) \ll 1$$

we may drop the higher order terms and conclude that the probability of deadlock is approximately the probability of cycles of length 2:

$$Nr^4/4R^2.$$

The conclusions from all this arithmetic are:

- The probability a transaction experiences deadlock is proportional to the degree of concurrency (N).

- The rate of deadlocks is proportional to $N^2$.

- The probability a waiting transaction deadlocks is not sensitive to the degree of concurrency.

These results have been observed in practice and in several analytic models but no convincing proof of the result are known [7].

## ISSUES IN DISTRIBUTED SYSTEMS

### Model of distributed system

A distributed system partitions the set of entities into disjoint sets called **nodes.** Transactions may execute at several nodes but at any instant, a transaction **resides** at a particular node. Initially a transaction resides at the node of its input message. In order for a transaction at node N1 to execute an action on an entity at node N2 the transaction must **migrate** to that node by executing the operation MIGRATE_TO(N2) at node N1 and then execute the operation MIGRATE_FROM(N1) at node N2. Each node participating in a transaction keeps a REDO and UNDO log for the transaction's actions on entities at that node.

In a distributed system, nodes may fail independently. This introduces a new kind of failure:

- **node restart:** for some reason all volatile entities at the node spontaneously change to null.

### Validity of serial history

Let T1,...,Tn be a set of transactions which execute on a system of m nodes. Each node has a history of the actions it executes, H1,...,Hm. To generalize the results for a single node system to a multi-node system we must exhibit a single schedule H such that:

- Each Hi is a subsequence of H and,
- Each Ti is a subsequence of H.

Among other things, the dependency set of H will be the same as the union of the dependency sets of H1,...,Hm. So H will be a single node history with the same who-gave-what-to-whom relation as the union of the Hi.

H may be demonstrated constructively by associating an initially zero counter with each node and with each transaction. Each time a node executes an action of transaction T the node and transaction counters are set to the maximum of the node and transaction counters plus 1. This counter is then associated with that action of the transaction. If all the actions are sorted major by their counter value and minor by their node index then the resulting ordering is a schedule H for all actions. This ordering has the desired properties [10,11].

### Reliability

Node restart in a distributed system is much like system restart in a single node system with the exception of transactions which have migrated among several nodes. Transactions which migrate complicate both node restart and transaction restart. The simplest approach to transaction restart is to adopt the rule [12]:

- Only the node of residence can initiate transaction restart.

This rule implies that when transaction, T, migrates from node, N1, node N1 abdicates the right to restart T. This in turn means that the MIGRATE_TO(N2) operation at node N1 must prevent a restart of node N1 from restarting T. Hence, as part of the MIGRATE_TO(N2) operation, node N1 must make a stable copy of the state of T. At a minimum, this state includes REDO and UNDO logs of transaction T along with all locks and the state of all volatile entities belonging to T (recall that T may migrate back to N1 and expect its program state to be preserved). Until the transaction commits or restarts, each node restart at N1 must reconstruct the state of T at node N1 from this information.

The commit operation broadcasts commit to each node participating in the transaction. The transaction restart operation broadcasts restart to each such participant. When all participants have acknowledged that they have performed their part of the commit or restart, the node of residence can terminate the transaction (commit) or reinitiate it (restart).

This commit protocol has the virtue of simplicity and may become the most commonly used algorithm. It has two properties which have caused a search for alternate algorithms. These two problems are:

- It requires a node to be able to record the entire state of a migrated transaction in stable storage.

- It prevents a node from unilaterally aborting a transaction which has migrated from that node. This may tie up the resources of one node for a long time if the transaction migrates to a node which subsequently fails.

The **two-phase commit** protocol is designed to eliminate the first problem and minimize the second. The two-phase commit protocol implements the commit action as follows: As part of the commit action, one participant of the transaction is appointed the **commit coordinator.**

The coordinator obeys the following protocol:

- Phase 1: Each participant of T is polled to see if it is prepared to commit.

- The coordinator enters phase 2 when it recoverably makes the decision to commit or abort.

    - If all participants agree to commit, the coordinator records the commit decision in T's stable REDO log and then broadcasts the commit message to each participant.

    - If any node does not agree to commit or does not respond within a time limit, the commit coordinator records the abort decision in T's stable UNDO log and then broadcasts the restart message to each participant.

    - The broadcast message is periodically re-broadcast to each participant until it acknowledges that it has acted on the message.

    - When the coordinator has received all the phase 2 acknowledgments it either terminates (commit) or reinitiates (restart) transaction T.

The participants obey the following protocol:

- Phase 0: Prior to agreeing to commit, any node may unilaterally undo all actions of the transaction at that node and broadcast transaction restart.

- Phase 1: Upon receiving a prepare to commit request,

  - If the node has unilaterally restarted T it responds with transaction restart,

  - Otherwise, the participant records the REDO and UNDO log of the transaction at the node in stable storage and responds with an agree to commit message.

- Phase 2: The participant then waits for the coordinator's decision.

  - If the coordinator broadcasts commit then the participant commits its part of T and then acknowledges completion to the coordinator.

  - If the coordinator broadcasts restart, then the participant undoes its part of T and then acknowledges completion to the coordinator.

The two-phase commit algorithm avoids saving the volatile parts of the transaction state. Node restart can restart any transaction not in phase two of the commit operation. Transactions having completed phase 1 and not completed phase 2 are called **in-doubt.** At node restart, the node must reestablish all X-mode locks belonging to in-doubt transactions as well as maintain the UNDO and REDO logs of such transactions until they resolved by the commit coordinator. The two-phase commit protocol also minimizes the period during which a node cannot unilaterally restart the transaction.

There are many variations of the two-phase commit protocol. There are almost no proofs about the properties of these protocols. The central theorem is:

THEOREM: If all participants observe the two-phase commit protocol then
either all participants eventually enter the commit state
without passing through the transaction restart state.
or all participants eventually enter the restart state
without passing through the commit state.
Lindsay [9] has the most careful presentation of this result.

## Concurrency

Concurrency is inherent in a distributed system (each node executes autonomously). The existence of a global history implies that the theorems about locking generalize to distributed systems. The node can acquire locks for actions on entities at that node. If all transactions are well-formed and two phase then the system will provide the illusion of a centralized serial history.

A problem with this approach is that each node has only a portion of the who-waits-for-whom graph. In order to detect deadlock cycles, someone must glue the pieces of the graph together. Otherwise, deadlock detection in a distributed system is analogous to deadlock detection in a centralized system.

## TRANSACTION CONCEPT IN A PROGRAMMING LANGUAGE

The transaction concept is a fundamental notion. It already appears in data definition and data manipulation languages associated with data management systems. It is likely to appear in more conventional languages in the future. This is a proposal for what such a language extension might include.

The language provides an abstraction for something like a **module type** which appears to the user as a collection of operations on entities. When a **module instance** is created it may be given the attributes **real, stable, recoverable volatile** or **volatile** which indicate whether REDO and UNDO records need to be kept for the instance and whether or not the actions must be deferred. If the instance is to be shared then it may be given the attribute **shared** which will cause the operations on the instance to acquire appropriate locks prior to manipulating the instance[1].

Each operation of the module must have corresponding undo and redo operations based on the UNDO and REDO log.

The language also supports the verbs COMMIT and ABORT which commit the transaction or undo it and commit it with an error message.

This is the view of the user of a module. The implementor of a module type needs to have an interface to a lock management facility which will handle lock requests and do deadlock detection. He also needs an interface to the log management facility which will accept log records and return them on request. Transaction undo and redo appears to the module as calls from recovery manager which invokes the undo and redo operation passing the the undo or redo log record [5,6].

---

[1] Note that monitors are inappropriate for the transaction notion. They violate the two phase lock protocol by releasing locks at procedure exit rather than at transaction commit.

## REFERENCES


[1] Davies, C.T., "Recovery Semantics for a DB/DC System," Proceedings ACM National Conference, 1973, pp. 136-141.

[2] Eswaran, K.E., J.N. Gray, R.A. Lorie, I.L. Traiger, "On the Notions of Consistency and Predicate Locks in a Relational Database System," Communications of the ACM, Vol. 19, No. 11, Nov. 1976, pp. 624-634.

[3] Gorski J., "A Formal Model for Transaction Back-up in a Database Environment", Institute of Informatics, Technical University of Gdansk, Poland, Draft, March 1979, pp. 13.

[4] Gray, J.N., R.A. Lorie, G.F. Putzolu, I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", Modeling in Data Base Management Systems. G.M. Nijssen editor, North Holland, 1976, pp. 365-394. Also IBM Research Report: RJ 1606.

[5] Gray, J.N., "Notes on Data Base Operating Systems", Operating Systems - An Advanced Course, R. Bayer, R.M. Graham, G. Seegmuller editors, Springer Verlag, 1978, pp. 393-481. Also IBM Research Report: RJ 2188, Feb. 1978.

[6] Gray J.N., P. McJones, M. W. Blasgen, R. A. Lorie, T. G. Price, G. F. Putzolu, I. L. Traiger, "The Recovery Manager of a Data Management System", IBM Research Report RJ 2623, August 1979. pp. 23.

[7] Korth H., Homan P. This is a brief discussion of work done by Hank Korth (of Stanford) and myself in 1978 and of discussions with Pete Homan (of IBM Hursley).

[8] Lampson B.W., H. E. Sturgis. "Crash Recovery in a Distributed Data Storage System", Xerox Research Report: ?, To appear in Communications of the ACM. April 1979, pp. 23.

[9] Lindsay B. G., P. G. Selinger, C. A. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, G. F. Putzolu, I. L. Traiger, B. W. Wade, "Notes on Distributed Databases", IBM Research Report: RJ 2571, July 1979. pp. 57.

[10] Papadimitriou C. H., "The Serializability of Concurrent Database Updates", Journal of the ACM, Vol. 26, No. 4, October 1979, pp. 631-653.

[11] Traiger I. L., Galtieri C. A., Gray J. N., Lindsay B. G., "Transactions and Consistency in Distributed Database Systems", To appear in ACM Transactions on Database Systems. also IBM Research Report: RJ 2555, June 1979, pp. 17.

[12] Rosenkrantz D. J., R. E. Sterns, R. E. Lewis, "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database Systems, Vol. 3, No. 2, June 1978, pp. 178-198.