RJ2555(33155)6/5/79
Computer Science

Research Report

TRANSACTIONS AND CONSISTENCY IN DISTRIBUTED DATABASE SYSTEMS

Irving L. Traiger
James N. Gray
Cesare A. Galtieri
Bruce G. Lindsay

IBM Research Laboratory
San Jose, California 95193

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

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication elsewhere and has been issued as a Research Report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely distributed until after the date of outside publication.

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

RJ2555(33155)6/5/79
Computer Science

TRANSACTIONS AND CONSISTENCY IN DISTRIBUTED DATABASE SYSTEMS

Irving L. Traiger
James N. Gray
Cesare A. Galtieri
Bruce G. Lindsay

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: The notions of transaction and of data consistency are defined for a distributed system. The cases of partitioned data, where a given file is stored only at one node, and replicated data, where the contents of a file may be replicated at several nodes are discussed It is argued that the distribution and replication of data should be transparent to the programs which use the data. Techniques for providing such transparency are abstracted and discussed.

By extending the notions of system schedule and system clock to handle multiple nodes, it is shown that consistency for partitioned data is amenable to the techniques used for centralized data.

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

TRANSACTIONS AND CONSISTENCY IN DISTRIBUTED DATABASE SYSTEMS

Irving L. Traiger,
James N. Gray,
Cesare A. Galtieri,
Bruce G. Lindsay

IBM Research Division
San Jose Laboratory
San Jose, California

ABSTRACT: The notions of transaction and of data consistency are defined for a distributed system. The cases of *partitioned data*, where a given file is stored only at one node, and *replicated data*, where the contents of a file may be replicated at several nodes are discussed. It is argued that the distribution and replication of data should be transparent to the programs which use the data. Techniques for providing such transparency are abstracted and discussed.

By extending the notions of system schedule and system clock to handle multiple nodes, it is shown that consistency for partitioned data is amenable to the techniques used for centralized data.

INTRODUCTION

To our knowledge, no general purpose distributed system provides the notion of ' network job ', a co-ordinated unit of work which operates at several nodes. Rather, a task which operates at several nodes must be carefully programmed to be sensitive to data location in the network and to node and network failures. It is our thesis that the difficulty of constructing such programs is a principal cause for the dearth of systems which do distributed processing.

We conjecture that the notion of transaction as used in most data management systems generalizes to the network environment. This paper suggests that network systems should provide the notion of a **transaction** as an abstraction which eases the construction of programs in a distributed system. Transactions would provide the programmer with the following types of transparency:

LOCATION TRANSPARENCY: Although data is geographically distributed and may move from place to place, the programmer can act as though all the data is in one node.

REPLICATION TRANSPARENCY: Although the same data item may be replicated at several nodes of the network, the programmer may treat the item as though it were stored as a single item at a single node.

CONCURRENCY TRANSPARENCY: Although the system runs many transactions concurrently, it appears to each transaction as though it is the only activity in the system. Alternately, it appears as though there is no concurrency in the system.

FAILURE TRANSPARENCY: Either all the actions of a transaction occur or none of them occur. Once a transaction occurs, its effects survive hardware and software failures.

Certainly, a system which provides transparency is as easy to use as a centralized system. If one writes a program to do something, the program will continue to work even though the data manipulated by the program is moved or replicated. The program may suffer a performance penalty if the data is remote, but if this is unacceptable the program or data may be moved together. These performance issues can be separated from the program logic.

The transaction notion is not a panacea. Rather, it is a convenience for a general class of applications. There are probably many applications which will be developed only when the application programmer can be relieved of concerns about failures, concurrency, location, and replication. Efficiency may dictate that some implementations be done in an application specific way, rather than using a general purpose transaction manager.

This paper is primarily concerned with transactions as an ideal or model of the highest levels of transparency and programmer convenience, and not necessarily a proposal for implementation techniques.

PRIOR ART

At present some systems provide some forms of transparency:

- IMS [1] requires that all data accessed by a transaction reside at a single node, hence it is a centralized database system. However, the Multiple Systems Coupling feature of IMS (MSC), provides location transparency for message handling among multiple IMS systems.

IMS also provides the transaction notion, and failure transparency. The program isolation feature of IMS comes very close to providing concurrency transparency. IMS has no notion of replicated data and so does not provide replication transparency. Most of the techniques IMS uses seem to generalize to distributed systems (as will be shown below).

- CICS 1.4 provides the transaction notion, location transparency, and failure transparency. CICS does not provide a lock manager so it does not provide concurrency transparency. Responsibility for concurrency control is delegated to the individual subsystems such as DL/1, TOTAL, ADABAS, or VSAM. However, it seems fairly clear that the lock manager of IMS or some other data management system could be generalized to operate in a CICS 1.4 network. CICS has no notion of replicated data.

- The SDD-1 system [3] provides both location and replication transparency allowing the user to think in terms of entities (files) rather than objects (fragments of files). However, SDD-1 does not have a general notion of transaction so it does not provide failure or concurrency transparency. (In SDD-1 a single Datalanguage statement is a transaction. In general an application requires several Datalanguage statements to perform an operation such as ' funds transfer ' or ' parts order ').

MODEL OF TRANSACTIONS IN A DISTRIBUTED SYSTEM

We assume that the system consists of a geographically dispersed collection of computers called **nodes** which are identified by unique node identifiers. We assume that nodes may be unavailable for a while (down) but that they eventually return to service.

All the nodes are connected together via a common **network.** This network carries messages from node to node. We assume that messages may be arbitrarily delayed but that each message is eventually delivered (perhaps not in the order sent).

The system supports a set of **entities** which are uniquely named. Each entity is represented within the system by one or more **objects** which are identified by <name,node> pairs, where name is the entity name and node is the ' place ' at which the object is located. At any instant, a **value** is associated with each object although this value may change with time. We have chosen this simple mapping for readability: entity E maps onto object <E,N> at node N. We believe that the results that follow generalize to more complex entity to object mappings (e.g. several objects at a node might represent the same entity, an object might represent part of an entity or multiple entities might map onto the same object.)

Files, records, message queues and terminals are examples of entities. All entities are assumed to be indivisible in the sense that if a file is an entity, its composite records are not considered entities and conversely, if records are considered entities then the containing file is not considered an entity. Further, objects representing the same entity at different nodes are considered to be independent of one another in the sense that they may (but probably should not) have different values.

If an entity is represented by multiple objects, then the entity is said to be **replicated.** An entity named E which is replicated at nodes N1,...,Nn is represented by the objects <E,N1>, ..., <E,Nn>. A system without replicas is called **partitioned** because each entity is at exactly one node (partition). If all objects reside at the same node, the system is called **centralized.**

Considering only *fully* replicated entities (where each replicated entity has a representitive object at each node) is another simplification of the model. We believe the following discussion generalizes to situations in which parts of an entity are represented by individual objects.

A transaction issues **requests** to manipulate entities. These requests against entities are translated by the system into one or more **actions** on objects. Actions are the primitives supported by the individual nodes of the network. Requests allow a location transparent view of objects.

For example, the user issues READ and WRITE *requests* against *entities* and the system translates these requests into a corresponding group of *actions* on *objects*. In particular, the translator keeps an entity-object directory which gives the node addresses of the objects representing each entity.

```
REQUEST ==>   TRANSLATOR   ==>   ACTIONS
  ON                               ON
ENTITY                          OBJECTS
```

Figure 1. The translation of requests to actions.

Each node provides a repertoire of *actions* which manipulate objects at that node (e.g. read or write record at that node, send or receive a message, etc.). However, we recognize only two generic actions:

READ_OBJ: examines but does not alter an object value. The occurrence of a read by transaction T of object <E,N> which has value Val is represented by:

<T,READ_OBJ,<E,N>,Val>

WRITE_OBJ: alters the value of an object independent of its prior value. The occurrence of a write by transaction T of object <E,N> to new value Val is represented by:

<T,WRITE_OBJ,<E,N>,Val>.

Each action operates only on *one* object and hence only at one node. There may be concurrency of execution in the network, but the actions at a node appear to 'happen' in some order. In particular, if two actions at a node are executed on the same object one will appear to 'happen' after the other.

In addition, the COMMIT_OBJ action is introduced to indicate the successful completion of the transaction. It is the last action of the transaction and acts on no particular object. The COMMIT_OBJ action of transaction T is represented by

<T,COMMIT_OBJ,-,->.

Reading or writing records or files fits this model nicely. Record entities are represented by record objects. All record objects are assumed to pre-exist with initially null values. Insertion of a record is modeled by writing a non-null value to a previously null object, and deletion is modeled by writing a null value to an object. More complex operations such as searching for records which satisfy some predicate or sending and receiving messages can be modeled as collections of actions.

One might generalize the request model to support more elaborate requests. For example associative (indexed or hashed) naming of entities or the support of entities which are fragmented among several nodes.

In general single requests (actions) are too primitive. Rather, requests (actions) are combined together (as a program) to form a transaction. Such a program, when executed produces a **transaction execution** or *transaction instance* which is a sequence of requests (actions) which must be viewed as a single logical unit of work. We assume that the transaction executes serially, completing one action before beginning the next. We will use the term transaction for the program and will ambiguously use the term transaction execution for a request sequence and for the corresponding action sequence. Where the distinction is important we will use the terms request sequence and action sequence.

One may think of transactions as complex operations on an abstract system state. In order to faithfully represent the intended application the system state, when quiesced, should always satisfy some application dependent consistency constraints among entity (object) values. We assume that transactions preserve these correctness criteria. They transform the system state from one consistent version to a new consistent version. During the transaction, the state may become inconsistent (e.g. in a funds transfer application, one bank account may be debited but another one not yet credited), but when a transaction commits the system state is again consistent.

Sample transactions from a banking application open accounts, transfer funds, pay interest, or mark an account ' held '. These transactions present the abstractions of account, money and client. For example the FundsTransfer transaction might have the form:

FundsTransfer;

```
READ      <input message A, B, DELTA>;
READ      ACCOUNT_BALANCE A; /* debit   */
WRITE     ACCOUNT_BALANCE A;
READ      ACCOUNT_BALANCE B; /* credit  */
WRITE     ACCOUNT_BALANCE B;
WRITE     <response message>;
COMMIT;
```

The actual transaction would be a (COBOL) program with computations interspersed with system requests. It would have symbolic (variable) names rather than entity names. We abstract this program by the sequence of requests (actions) it issues in a particular execution.

All actions on objects are performed at the node of the object. Whenever a node participates in a transaction execution, the node allocates an **agent** for the transaction execution. The agent keeps track of the local transaction execution state and performs read and write actions for the transaction at that node. Whenever a non-local action is requested by a transaction at a node, the node:

(1) requests that the object owner perform the action for the requesting transaction,
(2) waits for a response from the owner,
(3) reflects the response to the requesting transaction as though it were a local request.

The node owning the object:

(1) receives the request for the action on the object,
(2) eventually schedules the transaction agent to perform the action on the object for the transaction,
(3) reflects the response to the requesting node.

Thus the transaction is executed synchronously, completing one action (request) before issuing the next, and ultimately issuing a commit action. This model does not preclude an implementation in which the locus of control for a transaction execution migrates from node to node as the focus of activity changes. We simply find it convenient to imagine that all actions of a transaction are issued by one node.

LOCATION TRANSPARENCY AND REPLICATION TRANSPARENCY

Data is partitioned among nodes to distribute work, minimize message traffic and minimize response time. For example, the telephone book is partitioned by area code.

Centralized systems may replicate entities for reliability, availability, or performance reasons. Some computer systems replicate selected files so that no single media error causes data unavailability. Other systems keep the same data organized in different ways so that access to the data is inexpensive: e.g. hashing records on customer name in one case and by invoice number in another case.

In distributed systems both availability and performance arguments for replicated data are even more compelling. If the availability of an entity is required for certain applications then the entity may be replicated at several nodes. Replication can also improve performance if the cost of storing and maintaining (updating) the replica is less than the cost of accessing it remotely. A frequently used file might be replicated at each node to minimize message traffic and time delay in answering queries against the file. Telephone books, price lists, and other frequently used files are often replicated in this way.

Partitioning and replication may complicate programming. Programs which are sensitive to the location of objects they manipulate are quite complex. To give a simple example, suppose that the FundsTransfer program had different logic depending on whether the debited and credited accounts were local or not. There would be four cases (both local, one local, the other local, both remote). Either the program will have to handle these four cases separately or the system will have to provide location transparency. The complexity of locating each record and issuing the appropriate call to the appropriate node would dwarf the logic of the FundsTransfer program. Location transparency also allows the movement of objects without invalidating application programs which reference the corresponding entities. These are the arguments for location transparency. A system which supports location transparency would accept the FundsTransfer program in the form presented above and would translate the requests into actions at the appropriate nodes.

The argument for replication transparency is similar. We would like the freedom of moving and replicating entities without affecting program logic. Having the application program explicitly locate and update each replica of an entity would greatly complicate the program. It should be as though the program is dealing with a single copy of all entities at a single node.

A system providing location transparency and replication transparency allows the programmer to think in terms of entities. It hides issues of locating the objects which represent the entity and of maintaining consistency among the replicas of a single entity. It gives the impression of a single-node, single-copy system.

Location and replication transparency may be provided by a translator which transforms requests on entities into actions on objects. Perhaps the simplest translator operates as follows: Suppose the entity named E has representative objects at nodes N1, N2, ..., Nn. Then the request:

<T,READ,E,Val>

could translate to the action:

<T,READ_OBJ,<E,Ni>,Val>

for some Ni in N1,....Nn, and the request:

<T,WRITE,E,Val>

could translate to the actions:

<T,WRITE_OBJ,<E,N1>,Val>
<T,WRITE_OBJ,<E,N2>,Val>
.
.
.
<T,WRITE_OBJ,<E,Nn>,Val>

That is, a read request of an entity causes a READ_OBJ of any representative object and a write request of an entity causes a WRITE_OBJ to every representative object.

Further, the request

<T,COMMIT,-,->

could translate to

<T,COMMIT_OBJ,-,->.

(Other actions (e.g. unlocks) will be added to the commit request when we discuss deferred update and concurrency transparency.)

A system providing this function or an equivalent function makes reading or writing non-local or replicated data transparent to the program. From the perspective of a transaction making requests, the fact that objects representing an entity are remote or are replicated is transparent.

The translations described above are very simple. Many variations are possible. For example, some actions on remote data may be *deferred*. In particular it may be desirable to defer writes to remote objects and then batch the writes at some time prior to the commit action. Such a strategy might reduce message traffic.

The following is a request translator which gives transparency but tends to defer writes (presumably in order to 'batch' them): Each transaction execution carries a set of per-node lists of deferred writes. The request translator maintains the list of writes transparently to the program and programmer. Whenever a transaction requests a write, the writes to remote replicas of the entity are added to the transaction's list of deferred actions. Subsequent deferred writes by the transaction on the same object supersede earlier ones so the list carries at most one write per object. Reads of objects are done as before subject to the restriction that the transaction instance must apply any deferred write actions to an object before reading the object. After deferred writes are done they may be removed from the list. As part of the commit request, the transaction must apply all its deferred write actions.

In this paper we assume that all other actions of a transaction must be applied prior to commit (e.g. all replicas must be updated). Hence general rules for deferring actions are: Reads cannot be deferred because the read must return the current value of the named object. But a write action by transaction T may be deferred subject to the constraints: It should precede subsequent actions by T on the object. It must precede the COMMIT_OBJ action of T.

A later section will show that deferring writes does not create inconsistency for other transaction executions so long as all updated records are locked in exclusive mode and such locks are maintained until the writer commits.

The requirement that *all* replicas of an updated entity be accessible while the transaction instance is active may be unacceptable. If there are many replicas it may make it impossible to ever update the entity because some replica is always unavailable. Techniques to allow updates when only some replicas are accessible are beyond the scope of this paper (i.e. we assume all actions are installed prior to transaction commit).

TRANSACTION CONSISTENCY

Transaction execution is not instantaneous; it may involve reading slow (secondary) storage or conversing with remote nodes via slow communication lines. Hence, several transactions are usually executed in parallel as an economy which improves resource utilization (hardware and information). If concurrency introduces inconsistencies or makes the design of transactions substantially more complex, then it is probably a false economy.

To give examples of the anomalies that may arise from parallel execution of transactions, consider the concurrent execution of two instances of the FundsTransfer transaction above acting on the same bank account. Suppose the account originally has 100 dollars, and that one transaction wants to credit 10 dollars and the other wants to debit 40 dollars. If the transactions run one after the other the final balance will be 70 dollars (100+10-40). Yet if the transactions run concurrently, they may both update the 100 dollar balance to give a balance of 60 or 110 dollars. This is called the lost update problem. Another form of inconsistency arises from reading records while they are in flux. For example if someone else read account A and account B *during* the execution of the FundsTransfer transaction then the reader might see a situation in which money had 'disappeared' (A debited but B not yet credited). Such situations would be impossible if there were no concurrency.

These concurrency anomalies are very difficult to understand and guard against and therefore most transaction management systems *hide* concurrency by implementing a lock protocol which precludes such anomalies. They automatically generate lock actions as part of the translation of requests into actions. Three new actions are introduced:

LOCK_S: requests the designated object in share mode. The request is only granted (action completed) when no other transaction instance is granted the object in exclusive mode. A LOCK_S action by transaction T on object <E,N> is represented as:

<T,LOCK_S,<E,N>,->

LOCK_X: requests the designated object in exclusive mode. The request is only granted (action completed) when no other transaction instance is granted the object (in any mode). A LOCK_X action by transaction T on object <E,N> is represented as:

<T,LOCK_X,<E,N>,->

UNLOCK: releases the lock on a designated object. An UNLOCK action by transaction T on object <E,N> is represented as:

<T,UNLOCK,<E,N>,->

A transaction instance may lock the same object many times and in different modes. Once an entity is locked in exclusive mode, it remains locked in exclusive mode until it is unlocked. The unlock action releases all prior lock requests by the transaction instance for the object.

Requesting a lock which is unavailable may cause the lock action to wait and not be granted (occur) until the lock is available. Hence each lock action runs the risk of deadlock: one transaction instance waiting for another which in turn waits (perhaps indirectly) for the first. One cannot in general avoid deadlock. It seems to best to detect it (either algorithmically or via timeout) and to treat deadlocks as failures which cause some of the deadlocked transactions to be undone and preempted. In this paper we treat deadlocks like other errors, the transaction is reset and retries.

In order to describe lock protocols which preclude inconsistency, we introduce some terminology:

A transaction execution is **well formed** if:

READ_OBJ actions are always preceded by a LOCK_S for the object.

WRITE_OBJ actions are always preceded by a LOCK_X for the object.

Further the transaction execution is **two-phase** if:

after a transaction issues an unlock action, it never issues a lock action.

A precursor to this paper [4] proved that if all transaction instances are well formed and two-phase then there are no concurrency anomalies. In particular such a lock protocol prevents one transaction instance from reading or updating uncommitted writes of another transaction. (We shall state a variant of this result more formally below.)

In this paper we will use a stronger form of two-phase: all locks will be held to the very end of the transaction execution (rather than some unlocks occurring prior to the commit request). In particular, when we discuss concurrency transparency, the application program will never issue a lock or unlock actions or requests. Rather, READ and WRITE *requests* are translated to READ_OBJ and WRITE_OBJ *actions* preceded by LOCK_S or LOCK_X *actions* respectively, and the COMMIT *request* is translated to the appropriate UNLOCK *actions*. To motivate this observe that locks are set and held for several reasons:

(1) To stabilize objects which are read.

(2) To hide from other transaction instances uncommitted object values because they may be inconsistent.

(3) To hide from other transaction instances uncommitted object values because they may later be undone (this prevents transaction undo from cascading to other transactions).

Issue (3) was not discussed in [4]. In that paper it was shown that two-phase and well formed were necessary and sufficient conditions to prevent concurrency anomalies. By adopting the stronger definition of two-phase (locks held to commit request) the necessity property has been sacrificed. However the recovery issues justify the stronger definition of two-phase.

At this point, a *transaction execution* is represented as a sequence of <transaction_name,action,object,value> 4-tuples, each such item being one action of the transaction (READ_OBJ, WRITE_OBJ, LOCK_S, LOCK_X, UNLOCK, or COMMIT_OBJ). Transaction name is a unique transaction *instance* name.

The execution of a centralized (single node) system may be described by a schedule which tells the order in which the actions of the various transactions instances were executed. So a **schedule** for the set of transaction executions, T1,T2,...,Tn, is any sequence S=<...,<Ti,Ai,Oi,Vi>,...> such that:

- each Ti is a sub-sequence of S, and
- the length of S is the sum of the lengths of the Ti.

This means that S is some merging of the transaction executions which preserves the order within each transaction and which does not leave out any actions.

Two kinds of schedules are particularly interesting: serial schedules and legal schedules.
A **serial schedule** completes all actions of one transaction execution before beginning the next transaction. A serial schedule has no concurrency.
A **legal schedule** is one in which conflicting lock requests are not simultaneously granted. Schedule S is legal if for any transaction executions T1 and T2, and any object O and if

S=<...<T1,LOCK_X,O,->,...,<T2,LOCK_X,O,->,...>

or,

S=<...<T1,LOCK_X,O,->,...,<T2,LOCK_S,O,->,...>

or,

S=<...<T1,LOCK_S,O,->,...,<T2,LOCK_X,O,->,...>

and if T1 does not UNLOCK O in the middle ellipsis then T1=T2.
(That is, no two transactions can concurrently have the same object locked in conflicting modes.)

The essential ordering of actions in a particular schedule may be abstracted by a relation which shows 'who told what to whom'. This relation is called the dependency relation of a schedule and captures the essence of the schedule (other aspects of the schedule are arbitrary orderings of unrelated actions).

The **dependency set of schedule S, DEP(S),** is the ternary relation such that: for any distinct transactions T1 and T2 and any object O with value V, <T1,<O,V>,T2> is in DEP(S) if:

S=<...,<T1,WRITE_OBJ,O,V>,...,<T2,WRITE_OBJ,O,V'>,...> or
S=<...,<T1,WRITE_OBJ,O,V>,...,<T2,READ_OBJ ,O,V >,...> or
S=<...,<T1,READ_OBJ ,O,V>,...,<T2,WRITE_OBJ,O,V'>,...>

It is further required that no <T3,WRITE_OBJ,O,V'> actions occur in the middle ellipsis.

If two schedules have the same dependency set then they both give each transaction the same picture of the world. The 'inputs' to the transaction instance T are:

{<O,V> | <T',<O,V>,T> in DEP(S)}

and its 'outputs' are:

{<O,V> | <T,<O,V>,T'> in DEP(S).}

In a serial system each transaction instance inputs and outputs a unique value for each object it accesses. If a transaction inputs or outputs several different values for the same entity the system is executing in a perceptibly non-serial order.

Since the dependency set describes the 'inputs' and 'outputs' of each transaction in a schedule, two schedules are said to be **equivalent** if they both have the same dependency set.

Serial schedules have no concurrency and so have no anomalies due to concurrency. The concurrency anomalies described above (lost updates and inconsistent reads) only occur in non-serial schedules. Hence we define serial schedules and all schedules equivalent to serial schedules as **consistent schedules.**

We can now state and prove the following result:

ASSERTION 1: If {T1,T2,..,Tn} is a set of well formed and two-phase transaction executions, then any legal schedule for them is equivalent to a serial (one transaction at a time) execution.

Proof: We must prove that any legal schedule S for well formed and two-phase transactions is a equivalent to a serial schedule. For each transaction execution T define HAPPEN(T) to be the index of the first UNLOCK action of T in S. (If the first UNLOCK of T is the i'th action in schedule S then HAPPEN(T)=i.) This defines a 'happening' sequence among T1, T2, ...,Tn. Assume without loss of generality that:

$$HAPPEN(T1) < HAPPEN(T2) < ... < HAPPEN(Tn).$$

We will prove that the original schedule S is equivalent to a permuted schedule S' in which all actions of transaction T1 occur first:
S'= T1 S' where S' is S with all T1 actions removed.

Consider any subsequence of S

$$S=<...,<T',A',O',V'>,<T1,A1,O1,V1>,...>.$$

If T' -= T1 then we argue that the two actions commute to produce a new legal schedule with the same dependency relation as S.

$$S'=<...,<T1,A1,O1,V1>,<T',A',O',V'>,...>.$$

If A1 or A' is a COMMIT_OBJ action, then S' is legal and DEP(S)=DEP(S') by inspection.

If O' -= O1 then S' is legal and by inspection of the dependency relation, DEP(S)=DEP(S') so S and S' are equivalent and S' is legal.

If O' = O1 then we observe that neither A1 nor A' is a WRITE_OBJ or LOCK_X action because that would imply that both T1 and T' have concurrently locked O1 and each has locked it in exclusive mode (because both are well formed). This would violate the assumption that S is a legal schedule. So if O' = O1 then A' and A1 are either LOCK_S, UNLOCK or READ_OBJ actions. Suppose A' -= UNLOCK, then again by inspection of the definition of the dependency relation, A' and A1 commute in this case. Now consider the case A' = UNLOCK, then DEP(S)=DEP(S') but it is not clear that S' is legal. But if A' = UNLOCK then HAPPEN(T') is prior to that action. But HAPPEN(T1) < HAPPEN(T') and T1 is two-phase so the last LOCK action by T1 precedes the A' action of T'. Hence A1 -= LOCK_S and S' is indeed legal. So if T' -= T1 the two actions commute and incidentally the resulting schedule is

legal. Thus all T1's actions commute with any actions by other transactions T', which precede T1's actions. This allows all T1 actions to be factored to the beginning of the schedule.

By induction this same transformation may be applied to the rest of the schedule. Ultimately one obtains a schedule of the form T1,T2,...Tn which is equivalent to the original schedule S. QED

This proves that if a system execution produces a legal schedule, then the well formed and two phase lock protocol provides concurrency transparency. The fact that the actions of a centralized system do indeed form a schedule (one action completes at a time) is reasonably obvious. The next sections show that the execution of a distributed system also produces a schedule and hence (by Assertion 1) that locking produces consistency in a distributed system.

CONSISTENCY IN DISTRIBUTED SYSTEMS

We now generalize the previous concurrency result to a system consisting of several nodes connected by a communications network.

Recall that each object <E,N> resides entirely at the node named N. In order for a transaction to execute, it may have to act on entities at several nodes. The execution of a particular transaction instance is as follows: It begins at some node and then issues successive actions. If the object is local, the action is performed locally. If the object is remote, the transaction requests the remote node to perform the action on the entity. The transaction waits for the successful completion of the remote action to be signaled before beginning the next action.

The perception of each node is that at any instant it has several actions on local objects to be executed. It executes these actions in some serial order, hence it executes some sequence of actions called a **node schedule:** Sj=<...<Ti,Ai,<Ei,j>,Vi>...>. The execution of the system might be described by these node schedules. However to understand the execution of the system it is necessary to have a single schedule rather than a vector of several uncorrelated schedules. Given such a merger one could apply the work of the previous section to discuss consistency.

Since each node is running autonomously it is not obvious that these node schedules can always be merged into a single system schedule which includes each node schedule as a subsequence. In particular the time ordering implicit in the definition of HAPPEN and its use in the proof of Assertion 1 does not immediately generalize to a distributed system without a global clock.

ASSERTION 2: The execution of a distributed system of n independent nodes each of which executes actions on local objects at the request of a set of transaction instances may be modeled as the execution of a single node executing actions in some sequential order.

Proof: The proof proceeds by:

(1) Showing that each node can maintain a clock which is consistent with the clock of each transaction instance which does work at the node.

(2) Using these clocks, a global schedule for the transaction instances is constructed which preserves the ordering of the transaction and system clocks.

The argument begins by assuming that each node, N, has a clock, CLOCK(N), which is initialized arbitrarily. Each transaction instance, T, also has a clock CLOCK(T). When the transaction instance begins, the clock of the transaction's home node is incremented and the

transaction's clock is set to this node clock. The clocks are defined in the style of Lamport [5]. (Note that these clocks are used as a proof mechanism. They are not needed in an implementation.)

The manipulation of the clocks is controlled as follows:

When transaction T at node N performs an action on a local object then

$$\text{CLOCK(N)}:=\text{CLOCK(N)}+1$$
$$\text{CLOCK(T)}:=\text{CLOCK(N)}$$

and the action is said to **happen** at new CLOCK(T).

When a transaction T at node N requests an action on a remote object at node N' then the following sequence takes place:

A message is sent from N to N' of the form:

$$<\text{'DO'},\text{CLOCK(N)},\text{T},\text{A},\text{O},\text{Val}>.$$

Node N' on receiving the message sets its clock to

$$\text{CLOCK(N')}:=\text{MAX(CLOCK(N')},\text{CLOCK(N))}+1.$$

When node N' executes action A on object O for transaction T, then

$$\text{CLOCK(N')}:=\text{CLOCK(N')}+1$$

the action is said to *happen* at new CLOCK(N').

After node N' executes the action, it responds to node N with the message:

$$<\text{'DONE'},\text{CLOCK(N')},\text{T},\text{A},\text{O},\text{Val}>$$

When N receives such a message it sets:

$$\text{CLOCK(N)}:=\text{MAX(CLOCK(N)},\text{CLOCK(N'))}+1$$
$$\text{CLOCK(T)}:=\text{CLOCK(N)}.$$

Clearly

(1) Each successive action of a transaction happens at a later time (by the transaction clock).

(2) If two actions occur at the same node then their times correspond to the order in which they are executed at the node.

Consider any execution of the distributed system. The execution defines a particular happening time (node clock) for each action and also defines a unique set of node schedules S1, S2, ..., Sn. Sort all the actions from all node schedules ascending on happening time as the major order and node index as a minor order to get a sequence S. By (1) above, each transaction execution T is a subsequence of S and S contains all actions so S is a schedule for the set of transaction executions (in the sense of the previous section). Similarly, by (2) above each Si is a subse-

quence of S. Hence S is a linear order which is global to all nodes of the distributed system. QED

The above construction is quite terse. Note that much of the ordering is arbitrary. Any schedule for the same set of transaction execution which has each node schedule and each transaction as a subsequence would be equally good. So for example if all transactions executions are completely local any merging of the node schedules is an acceptable schedule.

Having shown that the behavior of a distributed system can be modeled by a single node schedule, one can apply previous results to show that:

ASSERTION 3: If all transaction executions are well formed and two-phase, then the execution of the transactions by a distributed system will be equivalent to some serial execution of the transactions by a system consisting of a single node.

Proof: The argument is identical to the argument for Assertion 1. By Assertion 2, any system execution corresponds to some schedule. Any such schedule will be legal since locks on object <E,N> are granted at node N and thus conflicting locks will not be granted concurrently. If the transaction executions are two-phase and well formed then the argument of Assertion 1 applies (note that the schedule gives each transaction instance a happening time). The schedule may therefore be permuted into an equivalent serial schedule. QED.

In a real system one might imagine that the activity of a transaction migrates from node to node as the transaction progresses. We have assumed that a single node controls the sequencing of the transaction execution. This assumption was purely for exposition and the results of the model generalize so long as the execution of the transaction remains serial (i.e. no parallelism within the transaction).

Assertions 1 and 3 are surprisingly powerful although they may seem rather innocuous. They imply that if a transaction makes a consistent transformation of the database state when it commits, and if the transaction is well formed and two phase then the transaction will not cause any inconsistencies for any other transaction. In particular, a transaction is free to defer updates to remote or replicated data (until commit) so long as it locks all updated objects and holds such locks to commit.

The fact the a transaction can read *any* replica of an entity so long as it updates *all* objects is also non-trivial. Certainly, if there are no other transactions executing, such a strategy will give consistent results. But based on assertions 1 and 3, so long as locks are set correctly, transactions appear to execute without concurrency. Thus any consistent strategy which works without concurrency, works with concurrency.

CONCURRENCY TRANSPARENCY

Summarizing the previous two sections: lock protocols exist which prevent concurrency anomalies. These protocols work for centralized and distributed systems. The lock protocol postulates that there is a lock manager at each node. Prior to reading or writing an object at that node, the local lock manager grants a LOCK_S or LOCK_X action for the transaction on the object.

Given these observations, concurrency transparency may be provided by automatically generating the required LOCK and UNLOCK actions for the program. This can be done as follows:

The case of partitioned data is simplest. Suppose that entity E is represented by the single object <E,N>. Then the request:

<T,READ,E,val>

is replaced by the actions:

<T,LOCK_S ,<E,N>,->
<T,READ_OBJ,<E,N>,Val>

and the request:

<T,WRITE,E,val>

is replaced by the actions:

<T,LOCK_X ,<E,N>,->
<T,WRITE_OBJ,<E,N>,val>

The case of replicated data is a generalization of the partitioned data case. Suppose entity E is represented by objects <E,N1>, ..., <E,Nn>. Then the request:

<T,READ,E,val>

is replaced by the actions:

<T,LOCK_S ,<E,N>,->
<T,READ_OBJ,<E,N>,Val>

for some node N in N1, ..., Nn, and the request:

<T,WRITE,E,val>

is replaced by the actions:

<T,LOCK_X ,<E,N1>,->
<T,WRITE_OBJ,<E,N1>,val>
<T,LOCK_X ,<E,N2>,->
<T,WRITE_OBJ,<E,N2>,val>
.
.
.
<T,LOCK_X ,<E,Nn>,->
<T,WRITE_OBJ,<E,Nn>,val>

The request:

<T,COMMIT,-,->

is replaced by the actions

<T,UNLOCK,O1,->
<T,UNLOCK,O2,->
.
.
.

<T,UNLOCK,On,->
<T,COMMIT_OBJ,-,->

for all objects O1, O2, ..., On locked by T.

This makes the lock actions implicit and transparently provides consistency.

FAILURE TRANSPARENCY

A variety of failures can prevent a transaction execution from completing successfully. Application detected anomalies (e.g., insufficient funds), database system difficulties (e.g., deadlock), and node failures (crashes) are familiar sources of trouble in single node systems. When multiple nodes are involved in a single transaction execution, communication network failures and distant node outages complicate the task of insuring system consistency.

Clearly, some of these failures will be visible to the end user, but it is possible to arrange the system so that the application program is insulated from many of these failures. Failure transparency relieves the application programmer of responsibility for restoring the system to a consistent state following a mid-transaction failure and for preserving the effects of the transaction execution in case of post-transaction failures.

Such a system must be able to deal with 1) application detected failures, 2) local node crashes, 3) communication network failures, and 4) failures originating at other nodes involved in the transaction execution. Different failures are detected by the system in different ways. An ABORT request is provided to allow the application to announce application failure. Local node crashes are detected during the node restart sequence. Network and distant node failures are detected by timeouts as well as by explicit notification from the net or distant node.

When an in-progress transaction fails, its effects are undone. Restoring the system state following the failure of a transaction requires remembering all actions which have been done up to the point of the failure. One commonly used strategy is for each node in the distributed system to maintain a *log* in which the information necessary to undo (and redo) the actions of the transaction's agent is recorded.

When a node fails, all transactions in-progress at the time of the failure will be undone as part of the node restart. The work of any committed transactions will be preserved (redone) as part of the restart of a node. If node N senses the failure of node N', then node N aborts any local uncommitted transaction executions (agents) involving node N'. The update strtegy allows reading but not updating an entity which has a replica at a failed node.

If transactions are to be atomic, the commit request which marks the end of a transaction execution must correspond to a single, atomic, recoverable *action* somewhere in the system (e.g. a log write to stable (magnetic) storage). Until a transaction commits, failures cause the transaction execution to be undone. After the transaction instance commits, all changes *at all nodes* must be retained (redone if necessary).

In a distributed system, commit must be carefully coordinated lest some nodes backup while others go forward. All nodes in a transaction execution must first agree not to abort the transaction on their own initiative. Then, after all have agreed to commit, some node can be the first to really commit (write the commit log record). This so-called, two-phase commit protocol has been described by several authors [6], [7], [8].

SUMMARY

We have described a very simple model of the execution of transactions in a distributed system. Based on this model, location, replication and concurrency transparency were defined and discussed.

Location, replication, concurrency and failure transparency ease application programming. The programmer is allowed to think in terms of entities rather than having to know the location(s) of the object(s) which represent them. Further the programmer is given read and write requests which in turn do sufficient locking and logging actions so that concurrency is hidden and failures are handled automatically (transaction is undone or redone). Lastly the commit and reset requests allow the programmer to control whether partially complete transactions are preserved or undone.

ACKNOWLEDGMENT

Bill Kent carefully read and criticized many drafts of this paper. He recommended many points of clarification and emphasis. We appreciate his help and patience.

REFERENCES

[1] IMS/VS General Information Manual, IBM form number GH20-1260, 1978.

[2] CICS General Information Manual, IBM form number GC33-0066, 1978.

[3] Rothnie J. B., Goodman N., Bernstein P. A., 'The Redundant Update Methodology of SDD-1: A System for Distributed Databases,' Computer Corporation of America, Report CCA-77-02, June 1977.

[4] Eswaran K. P., Gray J. N., Lorie R. A., Traiger I. L., 'On the Notions of Consistency and Predicate Locks in a Relational Database System,' CACM, Vol. 19, No. 11, November 1976.

[5] Lamport L., 'Time, Clocks, and the Ordering of Events in a Distributed System', CACM, Vol 21, No. 7, July 1978.

[6] Gray J.N., 'Notes on Database Operating Systems', *Operating Systems - An Advanced Course*, R. Bayer, R.M. Graham, G. Seegmuller editors, pp. 393-481, Springer Verlag, 1978.

[7] Lampson B. W., Sturgis H. E., 'Crash Recovery in Distributed Systems', Xerox Palo Alto Research Report, 1976. To appear in CACM.

[8] Rosenkrantz D. J., Stearns R. E., Lewis P.M., 'System Level Concurrency Control for Distributed Database Systems', ACM Transactions on Database Systems, Vol. 3, No. 2, June 1978.