Transactions and Consistency in Distributed Database Systems

IRVING L. TRAIGER, JIM GRAY, CESARE A. GALTIERI, and BRUCE G. LINDSAY
IBM Research Laboratory

The concepts of transaction and of data consistency are defined for a distributed system. The cases of partitioned data, where fragments of a file are stored at multiple nodes, and replicated data, where a file is 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. That is, the programming interface should provide location transparency, replica transparency, concurrency transparency, and failure transparency. Techniques for providing such transparencies are abstracted and discussed.

By extending the notions of system schedule and system clock to handle multiple nodes, it is shown that a distributed system can be modeled as a single sequential execution sequence. This model is then used to discuss simple techniques for implementing the various forms of transparency.

Categories and Subject Descriptors: H. 2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*

General Terms:

Additional Key Words and Phrases: Data replication, data partitioning, concurrency control, recovery

1. INTRODUCTION

To our knowledge, no general-purpose distributed system provides the notion of a "network job," a coordinated unit of work which operates at several nodes. Rather, a task that 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 of 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 transparencies.

(1) *Location Transparency.* Although data are geographically distributed and

Authors' present addresses: J. Gray, Tandem Computers Incorporated, 19333 Valico Parkway, Cupertino, CA 95014; I. L. Traiger, C. A. Galtieri, and B. G. Lindsay, IBM Research Laboratory, San Jose, CA 95193.

may move from place to place, the programmer can act as if all the data were in one node.

(2) *Replication Transparency.* Although the same data item may be replicated at several nodes of the network, the programmer may treat the item as if it were stored as a single item at a single node.

(3) *Concurrency Transparency.* Although the system runs many transactions concurrently, to each transaction it appears as if it were the only activity in the system. Alternatively, it appears as if there were no concurrency in the system.

(4) *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 that 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 are moved or replicated. The program may suffer a performance penalty if the data are 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 instead of by 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 is not necessarily a proposal for implementation techniques.

1.1 Prior Art

At present some systems provide some forms of transparency.

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

(2) The distributed INGRES system [11] provides location and replica transparency although, like SDD-1, it does not provide a notion of transaction. In INGRES, a single QUEL statement is a transaction. This implies that it does not provide either failure or concurrency transparency for transactions which are groups of QUEL statements.

(3) IMS [6] 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 for transactions within a single node. IMS has no notion of replicated or partitioned data and so does not provide replica or location transparency. It is shown below that most of the techniques IMS uses seem to generalize to distributed systems.

(4) CICS 1.4 [2] provides the transaction notion, location transparency, and failure transparency. Unfortunately, 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, System 2000, 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. Similarly, CICS has no notion of replicated data.

(5) Similar comments apply to Tandem's Encompass system, which supports location transparency, concurrency transparency, and failure transparency but not replica transparency [12].

In summary, currently available distributed systems provide very limited forms of transparency.

There have been many formal studies of these issues. The development presented here is closest to the work of Eswaran, Gray, Lorie, and Traiger [3] and to Rosenkrantz, Stearns, and Lewis [9, 10]. This paper differs from its predecessors in that it

(1) explicitly proves the existence of a global serial schedule (given the assumption that each transaction executes sequentially);

(2) introduces the terms location transparency, replica transparency, concurrency transparency, and failure transparency;

(3) develops a two-level model of entity → objects and request → actions which allows a clearer presentation of techniques for discussing the forms of transparency.

2. 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 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.

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. (See Figure 1.) We have chosen the simple mapping $E \rightarrow \langle E, N \rangle$ for readability. We believe that the results that follow generalize to more complex entity to object mappings. For example, several

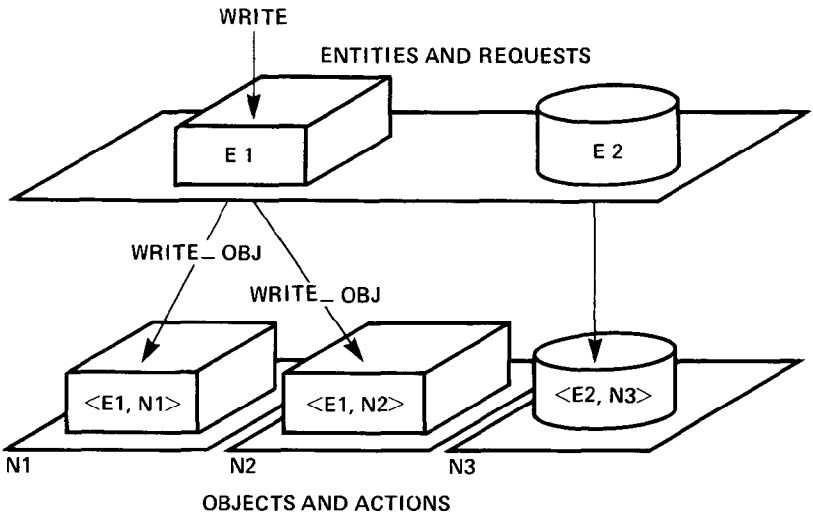

Fig. 1. Two levels of abstraction: The top level provides requests on entities (named E_1 and E_2). Entities are represented by one or more low-level objects ($\langle E_1, N_1 \rangle$ and $\langle E_1, N_2 \rangle$ are objects representing E_1 at nodes N_1 and N_2). Requests on entities are translated to one or more actions on the representative objects.

objects at a node might represent the same entity or an object might represent part of an entity.

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 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 $N_1, \ldots, N_n$ is represented by the objects $\langle E, N_1 \rangle, \ldots, \langle E, N_n \rangle$. 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 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 particular application associates a meaning with each entity (e.g., "entity E_1 represents bank account number 34532," "entity E_2 represents the assets of branch 39," ...). The collection of entities is presumed to satisfy some global constraint called the system *consistency constraint.* A fragment of such a constraint might be: "The assets of branch 39 is equal to the sum of the accounts at branch 39." We represent the constraint by the predicate C on entities and their values. The predicate C is generally not known to the system but is embodied in the structure of the transactions.

A transaction issues *requests* to manipulate entities. These requests against

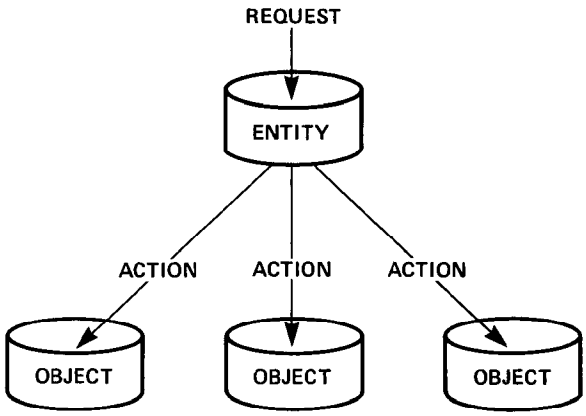

Fig. 2. The translation of requests to action.

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 transparent view of objects. (See Figure 2.)

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.

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

(1) READ_OBJ: examines but does not alter an object value. The occurrence of a read by transaction T of object $\langle E, N \rangle$ which has value Val is represented by

$$\langle T, \text{READ\_OBJ}, \langle E, N \rangle, \text{Val} \rangle.$$

(2) WRITE_OBJ: alters the value of an object independent of its prior value. The occurrence of a write by transaction T of object $\langle E, N \rangle$ to new value Val is represented by

$$\langle T, \text{WRITE\_OBJ}, \langle E, N \rangle, \text{Val} \rangle.$$

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

$$\langle T, \text{COMMIT\_OBJ}, \langle -, N \rangle, - \rangle.$$

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.

Reading or writing records or files fits this model nicely. Record entities are

represented by record objects. All record objects are assumed to preexist with initially null values. Insertion of a record is modeled by writing a nonnull 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. When executed, such a program produces a *transaction execution*, which is a sequence of requests (actions) that 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 use the term transaction ambiguously for the program and for a transaction execution considered as a request sequence or the corresponding action sequence. Where the distinction is important, we use the terms program, request sequence, and action sequence.

One may think of transactions as complex operations on the system state. We assume that transactions preserve the system consistency constraint. In particular, we assume that if the transaction T runs alone (i.e., without any other concurrent transactions), it transforms the system state from one consistent version to a new consistent version (the consistency constraint C is a precondition and a postcondition of the program T). 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 are: 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 that transaction. The agent keeps track of the local transaction state and performs read and write actions for the transaction at that node. Whenever a nonlocal

action is requested by a transaction at a node, the source node

(1) requests that the node owning the object perform the action for the requesting transaction,
(2) waits for a response from the node owning the object,
(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's 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 to each node visited. 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.

3. LOCATION TRANSPARENCY AND REPLICA TRANSPARENCY

Data are 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 read 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 that 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 would have to handle these four cases separately, or the system would 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 that reference

the corresponding entities. These are the arguments for location transparency. A system that 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 replica 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 if the program were dealing with a single copy of all entities at a single node.

A system providing location transparency and replica 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 replica 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 $N_1, N_2, \ldots, N_n$. Then the request

$$\langle T, \text{READ}, E, \text{Val} \rangle$$

could translate to the action

$$\langle T, \text{READ\_OBJ}, \langle E, N_i \rangle, \text{Val} \rangle$$

for some $N_i \in \{N_1, \ldots, N_n\}$, and the request

$$\langle T, \text{WRITE}, E, \text{Val} \rangle$$

could translate to the actions

$$\begin{aligned} &\langle T, \text{WRITE\_OBJ}, \langle E, N_1 \rangle, \text{Val} \rangle \\ &\langle T, \text{WRITE\_OBJ}, \langle E, N_2 \rangle, \text{Val} \rangle \\ &\qquad\vdots \\ &\langle T, \text{WRITE\_OBJ}, \langle E, N_n \rangle, \text{Val} \rangle. \end{aligned}$$

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

$$\langle T, \text{COMMIT}, -, - \rangle$$

could translate to the actions

$$\langle T, \text{COMMIT\_OBJ}, \langle -, N \rangle, - \rangle$$

for each node N at which transaction T performed an action. (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 nonlocal 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 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 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 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

(1) reads cannot be deferred because the read must return the current value of the named object;
(2) a write action by transaction T may be deferred subject to the constraints: it should precede subsequent actions by T on the object, and it must precede the COMMIT_OBJ action of T at that node.

A later section shows that deferring writes does not create inconsistency for other transactions 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 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). Gifford [4], using a model much like the one presented here, gives an update algorithm which tolerates the unavailability of a minority of the representatives of an entity.

4. 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.

As an example of the anomalies that may arise from parallel execution of transactions, consider the concurrent execution of two FundsTransfer transactions acting on the same bank account. Suppose the account 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.

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

$$\langle T, \text{LOCK\_S}, \langle E, N \rangle, - \rangle.$$

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

$$\langle T, \text{LOCK\_X}, \langle E, N \rangle, - \rangle.$$

(3) UNLOCK: releases the lock on a designated object. An UNLOCK action by transaction T on object $\langle E, N \rangle$ is represented as

$$\langle T, \text{UNLOCK}, \langle E, N \rangle, - \rangle.$$

A transaction 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 for the object.

Requesting a lock which is unavailable may cause the lock action to wait and not be granted until the lock is available. Hence each lock action runs the risk of deadlock: one transaction waiting for another which in turn waits (perhaps indirectly) for the first. One cannot in general avoid deadlock. It seems best to detect deadlock (either algorithmically or via time-out) 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's actions are undone and the transaction is restarted.

In order to describe the lock protocols which preclude inconsistency, we

introduce some terminology:

(1) A lock action on entity E is said to *cover* all subsequent actions by that transaction on entity E up to the next unlock of entity E by that transaction.
(2) A transaction execution is *well formed* if
READ_OBJ actions are always covered by a LOCK_S for the object;
WRITE_OBJ actions are always covered by a LOCK_X for the object.
(3) A transaction execution is *two-phase* if after a transaction issues an UNLOCK action, it never issues a lock action.

A precursor to this paper [3] proved that if all transactions are well formed and two-phase, then there are no concurrency anomalies. In particular, such a lock protocol prevents one transaction from reading or updating uncommitted writes of another transaction. (We state a variant of this result more formally below.)

In this paper we use a stronger form of two-phase: all locks will be held to the very end of the transaction execution (instead of some unlocks occurring prior to the commit request). In particular, when we discuss concurrency transparency, the application program will never issue lock or unlock 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* followed by COMMIT_OBJ *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 transactions uncommitted object values because they may be inconsistent;
(3) to hide from other transactions uncommitted object values because they may later be undone (this prevents transaction undo from cascading to other transactions).

Issue (3) was not discussed in [3]. In that paper it was shown that two-phase and well formed were necessary and sufficient conditions to prevent concurrency anomalies. By our adoption of 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.

To summarize, 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).

$$\langle \langle T, A_i, O_i, V_i \rangle \mid i = 1, \ldots, n \rangle.$$

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 were executed. So a *schedule for the set of transaction executions*, $T_1, T_2, \ldots, T_n$, is any sequence $S = \langle \ldots, \langle T_i, A_{ij}, O_{ij}, V_{ij} \rangle, \ldots \rangle$ such that each T_i is a subsequence of S, and the length of S is the sum of the lengths of the T_i. 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 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 transactions T_1 and T_2, and any object O if

$$S = \langle \ldots, \langle T_1, \text{LOCK\_X}, O, - \rangle, \ldots, \langle T_2, \text{LOCK\_X}, O, - \rangle, \ldots \rangle$$

or

$$S = \langle \ldots, \langle T_1, \text{LOCK\_X}, O, - \rangle, \ldots, \langle T_2, \text{LOCK\_S}, O, - \rangle, \ldots \rangle$$

or

$$S = \langle \ldots, \langle T_1, \text{LOCK\_S}, O, - \rangle, \ldots, \langle T_2, \text{LOCK\_X}, O, - \rangle, \ldots \rangle$$

and if T_1 does not UNLOCK O in the middle ellipsis then $T_1 = T_2$. (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 relation of schedule S, DEP(S)* is the ternary relation such that for any distinct transactions T_1 and T_2 and any object O with value V, then $\langle T_1, \langle O, V \rangle, T_2 \rangle \in \text{DEP}(S)$ if

$$S = \langle \ldots, \langle T_1, \text{WRITE\_OBJ}, O, V \rangle, \ldots, \langle T_2, \text{WRITE\_OBJ}, O, V_1 \rangle, \cdots \rangle$$

or

$$S = \langle \ldots, \langle T_1, \text{WRITE\_OBJ}, O, V \rangle, \ldots, \langle T_2, \text{READ\_OBJ}, O, V \rangle, \ldots \rangle$$

or

$$S = \langle \ldots, \langle T_1, \text{READ\_OBJ}, O, V \rangle, \ldots, \langle T_2, \text{WRITE\_OBJ}, O, V_1 \rangle, \ldots \rangle.$$

It is further required that for all T_3 no $\langle T_3, \text{WRITE\_OBJ}, O, V_2 \rangle$ actions occur in the middle ellipsis. In addition, DEP(S) is augmented by two transactions INITIAL and FINAL so that if

$$S = \langle \ldots, \langle T, \text{READ\_OBJ}, O, V \rangle, \ldots \rangle$$

or

$$S = \langle \ldots, \langle T, \text{WRITE\_OBJ}, O, V \rangle, \ldots \rangle$$

then

$$\langle \text{INITIAL}, \langle O, V \rangle, T \rangle \in \text{DEP}(S)$$

if the prior actions do not include the action $\langle T_i, \text{READ\_OBJ}, O, V_i \rangle$

and

$$\langle T, \langle O, V\rangle, \text{FINAL}\rangle \in \text{DEP}(S)$$

if the later actions do not include the action $\langle T_i, \text{WRITE\_OBJ}, O, V_i\rangle$.

If two schedules have the same dependency relation, then they both give each transaction the same inputs and outputs. The "inputs" to the transaction T are

$$\{\langle O, V\rangle \mid \langle T', \langle O, V\rangle, T\rangle \in \text{DEP}(S)\}$$

and its "outputs" are

$$\{\langle O, V\rangle \mid \langle T, \langle O, V\rangle, T'\rangle \in \text{DEP}(S)\}.$$

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

Since the dependency relation describes the "inputs" and "outputs" of each transaction in a schedule, we define two schedules to be *equivalent* if they both have the same dependency relation.

Serial schedules have no concurrency and so have no anomalies due to concurrency. The concurrency anomalies described above (lost updates and inconsistent reads) occur only in nonserial 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 [3, Assertion 8a; 9, Theorem 1]. *If $\{T_1, T_2, \ldots, T_n\}$ is a set of well-formed and two-phase transactions, 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 equivalent to a serial schedule. For each transaction T, define HAPPEN(T) to be the index in S of the first UNLOCK action of T in S. (If the first UNLOCK of T is the ith action in schedule S, then HAPPEN(T) = i.) This defines a "happening" sequence among $T_1, T_2, \ldots, T_n$. Assume without loss of generality that

$$\text{HAPPEN}(T_1) < \text{HAPPEN}(T_2) < \cdots < \text{HAPPEN}(T_n).$$

We will prove that the original schedule S is equivalent to a permuted schedule S_1 in which all actions of transaction T_1 occur first:

$$S_1 = T_1 S_2 \text{ where } S_2 \text{ is } S \text{ with all } T_1 \text{ actions removed.}$$

Consider any subsequence of S.

$$S = \langle \ldots, \langle T_i, A_i, O_i, V_i\rangle, \langle T_1, A_1, O_1, V_1\rangle, \ldots\rangle.$$

If $T_i \neq T_1$, then we argue that the two actions commute to produce a new legal schedule with the same dependency relation of S.

$$S' = \langle \ldots, \langle T_1, A_1, O_1, V_1\rangle, \langle T_i, A_i, O_i, V_i\rangle, \ldots\rangle.$$

If A_1 or A_i is a COMMIT_OBJ action, then S' is legal and DEP(S) = DEP(S') by inspection.

If $O_i \neq O_1$, 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 $O_i = O_1$, then we observe that neither A_1 nor A_i is a WRITE_OBJ or LOCK_X action because that would imply that both T_1 and T_i have concurrently locked O_1 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 $O_i = O_1$, then A_i and A_1 are either LOCK_S, UNLOCK, or READ_OBJ actions. Suppose $O_i = O_1$ and $A_i \neq$ UNLOCK. Then again by inspection of the definition of the dependency relation, A_i and A_1 commute in this case.

Now consider the case $O_i = O_1$ and A_i = UNLOCK. Then DEP(S) = DEP(S') because lock actions do not enter into the definition of the dependency relation. But it is not clear that S' is legal. If A_1 is UNLOCK or READ_OBJ, then S' is obviously legal if S is legal. The difficult case is A_1 = LOCK_S (it was pointed out above that LOCK_S, UNLOCK, and READ_OBJ are the only possibilities). Let A_i be the jth action of S. If A_i = UNLOCK, then since T_i is two-phase, HAPPEN(T_i) $\leq j$. Similarly if A_i = LOCK_S, then since T_1 is two-phase, $j <$ HAPPEN(T_1). So if A_i = UNLOCK, HAPPEN(T_i) $<$ HAPPEN(T_1). But we assumed HAPPEN(T_1) $<$ HAPPEN(T_i). This contradiction shows that S' is legal if A_i = UNLOCK.

This exhausts all the cases and shows that if $T_i \neq T_1$, the two actions commute and, incidentally, the resulting schedule is legal. Thus all T_1's actions commute with any actions by other transactions T_i, which precede T_1's actions. This allows all T_1 actions to be factored to the beginning of the schedule.

By induction, this same transformation may be applied to the rest of the schedule. Inductively one obtains a schedule of the form $T_1, T_2, \ldots, T_n$, which is equivalent to the original schedule S. □

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 if transactions execute operations sequentially, the execution of a distributed system also produces a schedule and hence (by Assertion 1) locking guarantees consistency in a distributed system.

5. 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 $\langle E, N \rangle$ resides entirely at the node named N. In order for a transaction to execute, it may have to act on objects at several nodes. The execution of a particular transaction 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 object. The transaction waits for the successful completion of the remote action to be signaled before beginning the next action. Thus, the execution of a transaction, T, in a distributed system may be modeled as

$$\langle \langle T, A_i, O_i, V_i \rangle \mid i = 1, \ldots, n \rangle.$$

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*:

$$S_j = \langle \ldots, \langle T_i, A_{ij}, O_{ij}, V_{ij} \rangle, \ldots \rangle.$$

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 sections to discuss consistency and transparency.

Since each node is running autonomously, it it 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. Indeed, if one allows concurrency within a transaction execution, then there may not be such a global schedule. On the other hand, we show below that if transactions execute one step at a time, then one may merge the local schedules to form a global schedule.

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 sequentially executing transactions, may be modeled as the execution of a single node executing actions in some sequential order.*

More formally, if $S_1, \ldots, S_n$ *are node schedules for transactions* $T_1, \ldots, T_m$, *then there is a global schedule S such that*

(*a*) *the* T_i *and* S_i *are subsequences of S, and*
(*b*) *if each* S_i *is legal, S is legal.*

PROOF. The proof of 2(a) proceeds in two steps:

(1) First, we show that each node can maintain a clock that is consistent with the clock of each transaction that does work at the node.
(2) Second, we use these clocks to construct a global schedule for the transaction 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, T, also has a clock CLOCK(T). When the transaction 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 [7]. (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_1, then the following sequence takes place: A message is sent from N to N_1 of the form

$$\langle \text{"DO"}, \text{CLOCK}(N), T, A, O, \text{Val} \rangle.$$

Node N_1, on receiving the message from node N, sets its clock to

$$\text{CLOCK}(N_1) := \text{MAX}(\text{CLOCK}(N_1), \text{CLOCK}(N)) + 1.$$

When node N_1 executes action A on object O for transaction T, then

$$\text{CLOCK}(N_1) := \text{CLOCK}(N_1) + 1;$$

the action is said to *happen* at new CLOCK(N_1). After node N_1 executes the action, it responds to node N with the message:

$$\langle \text{"DONE"}, \text{CLOCK}(N_1), T, A, O, \text{Val} \rangle.$$

When N receives such a message, it sets

$$\text{CLOCK}(N) := \text{MAX}(\text{CLOCK}(N), \textit{CLOCK}(N_1)) + 1$$

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

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 $S_1, S_2, \ldots, S_n$. 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 S_i is a subsequence of S. Hence, S is a linear order which is global to all nodes of the distributed system.

This establishes Assertion 2(a).

To establish Assertion 2(b), consider any object O at node N acted upon by schedule S. Let S_i be the node schedule of S at node N. All lock and unlock actions on O in S appear in the subsequence S_i which is legal. Hence, S is legal with respect to every object O.

This establishes 2(b). □

The above construction is quite terse. Note that much of the ordering is arbitrary. Any schedule for the same set of transactions which has each node schedule and each transaction as a subsequence would be equally good. So, for example, if all transaction 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 any legal 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 legal system execution corresponds to some legal schedule. Any such schedule will be legal since locks on object $\langle E, N \rangle$ 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 a happening time). The schedule may therefore be permuted into an equivalent serial schedule. □

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 that a transaction can read *any* replica of an entity so long as it updates *all* replicas is also nontrivial. Certainly, if there are no other transactions executing, such a strategy will give consistent results. But on the basis of 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.

6. CONCURRENCY TRANSPARENCY

To summarize the previous two sections, lock protocols exist which prevent concurrency anomalies. These protocols work for centralized and distributed systems. The protocols postulate that there is a lock manager at each node. Prior to reading or writing an object at that node, the transaction executes a LOCK S or LOCK X action on the object. The lock manager will delay the completion of the LOCK operation while the lock is granted to another transaction in a conflicting lock mode.

Given these observations, concurrency transparency may be provided by automatically generating the required LOCK and UNLOCK actions as part of the execution of a request. This automatic locking frees the programmer from having

to issue explicit lock requests. This can be done as follows:

The case of partitioned data is simplest. Suppose that entity E is represented by the single object $\langle E, N \rangle$. Then the request $\langle T, \text{READ}, E, \text{Val} \rangle$ is replaced by the actions

$$\langle T, \text{LOCK\_S}, \langle E, N \rangle, - \rangle$$
$$\langle T, \text{READ\_OBJ}, \langle E, N \rangle, \text{Val} \rangle$$

and the request $\langle T, \text{WRITE}, E, \text{Val} \rangle$ is replaced by the actions

$$\langle T, \text{LOCK\_X}, \langle E, N \rangle, - \rangle$$
$$\langle T, \text{WRITE\_OBJ}, \langle E, N \rangle, \text{Val} \rangle.$$

The case of replicated data is a generalization of the partitioned data case. Suppose entity E is represented by objects $\langle E, N_1 \rangle, \ldots, \langle E, N_n \rangle$. Then the request $\langle T, \text{READ}, E, \text{Val} \rangle$ is replaced by the actions

$$\langle T, \text{LOCK\_S}, \langle E, N \rangle, - \rangle$$
$$\langle T, \text{READ\_OBJ}, \langle E, N \rangle, \text{Val} \rangle$$

for some node $N \in \{N_1, \ldots, N_n\}$, and the request $\langle T, \text{WRITE}, E, \text{Val} \rangle$ is replaced by the actions

$$\begin{aligned}
&\langle T, \text{LOCK_X}, && \langle E, N_1 \rangle, - \rangle \\
&\langle T, \text{WRITE_OBJ}, && \langle E, N_1 \rangle, \text{Val} \rangle \\
&\langle T, \text{LOCK_X}, && \langle E, N_2 \rangle, - \rangle \\
&\langle T, \text{WRITE_OBJ}, && \langle E, N_2 \rangle, \text{Val} \rangle \\
&&\vdots \\
&\langle T, \text{LOCK_X}, && \langle E, N_n \rangle, - \rangle \\
&\langle T, \text{WRITE_OBJ}, && \langle E, N_n \rangle, \text{Val} \rangle.
\end{aligned}$$

The request $\langle T, \text{COMMIT}, -, - \rangle$ is replaced by the actions

$$\begin{aligned}
&\langle T, \text{UNLOCK}, && O_1, - \rangle \\
&\langle T, \text{UNLOCK}, && O_2, - \rangle \\
&&\vdots \\
&\langle T, \text{UNLOCK}, && O_n, - \rangle \\
&\langle T, \text{COMMIT_OBJ}, && \langle N_1, - \rangle, - \rangle \\
&&\vdots \\
&\langle T, \text{COMMIT_OBJ}, && \langle N_m, - \rangle, - \rangle
\end{aligned}$$

for all objects $O_1, O_2, \ldots, O_n$ locked by T and all nodes $N_1, \ldots, N_m$ which performed actions of T.

This makes the lock actions implicit and transparently provides consistency.

7. 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 ensuring 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 midtransaction failure and for preserving the effects of the transaction execution in case of postcommit 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 time-outs 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 that 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'.

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 executes this commit action, failures cause the transaction execution to be undone. After the transaction executes this commit action, 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 nodes 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 [7, 8, 9].

8. SUMMARY

We have described a very simple model of the execution of transactions in a distributed system. On the basis of this model, location, replica, concurrency, and failure transparencies have been defined and discussed.

Location, replica, concurrency, and failure transparencies 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 represents 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). It has been shown that these locking protocols provide concurrency transparency. Last, the commit and abort requests allow the programmer to control whether partially complete transactions are preserved or undone.

ACKNOWLEDGMENTS

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. The referees also made several constructive suggestions about the paper.

REFERENCES

1. BERNSTEIN, P.A., SHIPMAN, D.W., GOODMAN, N., AND ROTHNIE, J.B. Concurrency control in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5,* 1 (March 1980), 1–17.
2. CICS *General Information Manual.* IBM form number GC33-0066, IBM, White Plains, N.Y., 1978.
3. ESWARAN, K.P., GRAY, J.N., LORIE, R.A., AND TRAIGER, I.L. The notions of consistency and predicate locks in a database system. *Commun. ACM, 19,* 11 (November 1976) 624–633.
4. GIFFORD, D.K. Weighted voting for replicated data. In *Proc. 7th Symp. Operating Systems Principles* (Pacific Grove, Calif., Dec.10–12), ACM, New York, 1979.
5. GRAY, J.N. Notes on database operating systems. In *Operating Systems—An Advanced Course,* R. Bayer, R.M. Graham, and G. Seegmuller, Eds. Springer-Verlag, New York, 1978.
6. IMS/VS. *General Information Manual.* IBM form number GH20-1260, IBM, White Plains, N.Y., 1978.
7. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM, 21,* 7 (July 1978) 558–565.
8. LAMPSON, B.W., AND STURGIS, H.E. Crash recovery in distributed systems. Xerox Palo Alto Research Rep., 1976. Xerox Parc, Palo Alto, Calif., 1976.
9. ROSENKRANTZ, D.J., STEARNS, R.E., AND LEWIS, P.M. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3,* 2 (June 1978), 178–198.
10. STEARNS, R.E., LEWIS, P.M., AND ROSENKRANTZ, D.J. Concurrency control for database systems. In *Proc. 17th Symp. Foundations of Computer Science,* ACM, New York, 1976.
11. STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng. SE-5,* 3. (May 1979), 188–194.
12. BORR, A. Transaction monitoring in encompass-reliable distributed transaction processing. In *Proc. 7th Int. Conf. Very Large Databases,* IEEE, New York, 1981.

Received May 1979; revised May 1981; accepted May 1981