M.I.T. LABORATORY FOR COMPUTER SCIENCE

May 7, 1979

Computer Systems Research Division

Request for Comments No. 173

BUILDING RELIABLE DISTRIBUTED SYSTEMS:

THE PROBLEM OF ATOMIC OPERATIONS

by Liba Svobodova

Attached is the paper that I have submitted to the 7th SOSP.

Abstract

The goal of this paper is to tie together recent results and insights that have emerged from the body of research on the update problem in distributed systems. Its intent is not to propose yet another algorithm. Rather, the paper seeks to present a realistic picture of where we stand, and what problems we should concentrate on in order to make distributed systems practical.

Introduction

Much of the work on distributed systems has concentrated on the problem of performing an update that involves several physical nodes. While not all distributed applications will require such rigorous control as is implied by the protocols that have emerged from this body of work, mechanisms for performing distributed updates atomically belong among the basic mechanisms of a distributed operating system. And, most of the reliability issues concerning distributed systems have been raised in connection with such mechanisms.

The goal of this paper is to analyze and tie together the recent advances and insights concerning the problem of modifying a set of objects in a distributed system in a single atomic operation. Three separate research reports recently came out of the Computer Systems Research Division of the Laboratory for Computer Science at M.I.T.: the work of Reed [REED78], Montgomery [MONT78] and Takagi [TAKA78]. While each of them presents a new approach to ensuring atomicity of concurrent transactions, what is particularly appealing and important about these reports is that they address the problem of implementation and develop innovative mechanisms that make the implementation of the protocols on a real distributed system feasible. These

three reports together, analyzed and compared, provide a number of interesting insights that represent a significant step towards our understanding of how to build reliable distributed systems.* Of course, the research presented in the three specified reports cannot be addressed in isolation from the earlier work in this area, in particular since all three pieces of research have a common root, the two-phase commit protocol developed independently for System R at IBM [GRAY78] and Lampson and Sturgis at Xerox PARC [LAMP76].

The paper extends the discussion of the atomic distributed update to the replicated database case and points out the essential differences between it and the partitioned database case. A revision of the general model of transaction and the assumptions regarding the properties of the underlying (hardware and software) system concludes the paper.

Atomic Operations

The distributed update problem can be divided into two categories:

1) update of a partitioned database, and

2) update of a replicated database.

It should be pointed out that a database is used here as a generalization, an abstraction, that is, it does not necessarily imply the kind of database to which the user interfaces through an elaborate database management system. The state information of any system (application) represents a database. More specifically, a partitioned database can be

* Although many of the problems that will be discussed here occur in a conventional system built around a central processor, the physical distribution and decentralization of control constrain the possible solutions. In particular, it is necessary to limit the number of messages that must be exchanged among the participating nodes to ensure reasonable performance. Another problem is that if one node sends requests to several different nodes, these requests will experience different delays, and consequently may be received in a different order relative to requests from other nodes.

defined as an abstract object that consists of several separately managed pieces (abstract objects) that may reside at different physical nodes. A replicated database is an abstract object that is implemented as a set of objects, called here images, where each image represents the same abstraction. In both cases, the problem is to ensure consistency among the different objects that form the representation. In the first case, the consistency constraints are defined by the semantics of the data; this is usually referred to as internal consistency. In the second case, it is usually required that all images are always the same; this is called the mutual consistency requirement. In both cases, it is unreasonable to require that the consistency constraints will hold at every instant; practical update algorithms must be allowed to produce brief inconsistencies as part of changing the whole database to a new consistent state. However, if all actions that need to be performed to step a database from one consistent state to another are grouped together in a single atomic operation, the consistency constraints are guaranteed to hold between these atomic operations. In the context of a partitioned database, atomic operations are usually called transactions; in this paper, these two terms will be used interchangeably.

The definition of an atomic operation is that it is indivisible; the temporal inconsistency of the database on which it operates is hidden within the operation. This indivisibility has two implications:

1) the results of an atomic operation are not affected by other, concurrent operations, and
2) an atomic operation is either carried to its completion, according to its specification or, if it fails, or if the originator decides

to abort it, it leaves the system in the state it was prior to the invocation of that operation.*

This paper is concerned mainly with the second aspect, that is, recoverability of an operation. Actually, as it stands, 2) also states a requirement of correctness That is, it requires that the program that implements an atomic operation either be provably correct or that there exist sufficient defenses in the system that will detect not just hardware failures and synchronization errors but also errors caused by the program that implements the operation. The first is an area of much current research; the second approach has been investigated by a group from the University of Newcastle upon Tyne [RAND78]. However, in addition to executing correctly in a standalone mode, an atomic operation must execute correctly in the presence of concurrent operations.*

This paper is not concerned with how errors are detected; it concentrates on the problem of recovery from those errors (failures) that are detectable. The class of detectable errors is assumed to include scheduling anomalies (e.g., deadlocks). Alto, an operation may be aborted (and thus must be recovered) at the request of its initiator.

The protocols for managing transactions in a distributed environment use two kinds of mechanisms:

a) mechanisms that enforce the correct order of individual actions, and

b) mechanisms that supervise commitment of the entire operation (transaction).

---

* In order for operations to be recoverable, scheduling of concurrent operations has to be handled carefully; thus these two aspects of indivisibility cannot be completely separated.

* In most schemes, this is guaranteed only if the other (concurrent) operations are also atomic. Reed's work is a notable exception [REED78].

Although it would be possible to design both a and b type mechanisms so that they could be used to build atomic operations for any kind of distributed database, it makes sense, as explained later in this paper, to make the mechanisms for partitioned databases somewhat different from those for replicated databases. In particular, for partitioned databases, the ordering constraints on actions that belong to different transactions need to be somewhat relaxed, while for replicated databases, the commitment constraints need to be relaxed. Therefore, the two types of distributed update will be treated separately. The following section analyzes the protocols for transactions that operate on a partitioned database. The replicated database case is discussed in the following section.

Protocols for Partitioned Databases

Transactions that operate on a partitioned database usually are modeled as having a central coordinator (a transaction manager, abbreviated TM) that issues requests to a selected set of entities (participants of a transaction) that manage different pieces of a database (database managers, or DMs). The updates of objects controlled by different DMs are coordinated by some variant of a two-phase commit protocol.*

Figure 1 is a model** of what happens in each DM for each transaction in which the DM participates. First, the TM sends a read request to the DM.

---

* It should be noted that it is not necessary to arrange the participants of an atomic transaction in such a two level hierarchy. In particular, two-phase commit protocols have been developed or at least shown to be possible for a multi-level hierarchy [REED78], daisy-chain [GRAY78, TRAI78] and even a general graph [GRAY79].

** In general, the requests sent to DMs in the first phase can be of the form "perform operation X", where a significant amount of work may need to be done by the recipient DM. However, most proposed schemes assume explicit read and write requests.

This may be followed immediately or later by a write request for the same object. Although the DM has the new values now, it does not make any irrevocable changes at this point because it is not yet clear that all participants will be willing or able to act on their part of the transaction. If any request cannot be performed, the whole transaction fails. If all requests succeeded in all involved DMs, the TM sends a prepare message to the DMs. The prepare message is the end of the first phase. Any DM that replies with a positive acknowledgement to the prepare message enters the second phase where it is committed to obey the decision of the TM, that is, it must either make the requested changes definite if the decision is "commit" or completely undo any changes made on the behalf of this transaction if the decision is "abort". Until it acknowledges the prepare message, each DM is free to refuse to perform the requested operations or cancel them on its own will.

Each phase has its own problems. The problem of how to schedule actions that are part of different (concurrent) operations must be resolved during the first phase. In this phase, the individual DMs can proceed independently. In the second phase, a careful coordination of all participants is necessary to ensure that either all of them commit or all of them abort their transaction. In addition, it is necessary to decide when the objects modified by one transaction can be made visible to other transactions. The choices are: right after the new values have been received (i.e. after the write request has been processed), after the prepare request has been received, or when the commit request is received.

Phase 1: Scheduling

The phase 1 activities of concurrent transactions can be coordinated in several different ways:

1) locking: a) centralized [MENA78], b) distributed with centralized deadlock detection [STON78], and c) distributed [GRAY78],

2) timestamp-based scheduling of accesses [TAKA78, REED78],

3) organization of the database that enforces correct ordering of messages [MONT78].

Locking schemes, unless some additional sequencing information is used, are all vulnerable to a deadlock. Deadlock detection in a distributed system may be an expensive proposition, unless locking is centralized (e.g., all requests for locks go through a centralized controller) in which case locking itself might be too expensive because of this extra step of having to acquire the locks.* Stonebraker [STON78] uses a compromise scheme where only the requests for locks that cannot be granted are reported to a central authority (the SNOOP).

Takagi provides the following insight: the schemes that do not use explicit locking are "...based on the observation that a consistent schedule of transactions is merely a (proper) sequencing of actions performed on the underlying objects..." [TAKA78]. Since these schemes are set up so that the proper ordering of actions can be resolved by each separate DM individually, they are free of deadlock. Timestamp-based schedules vary in how they handle out of order (outdated) requests. One approach is to discard a delayed (older) request (and consequently the transaction that generated that request) if a newer request (that is, a request with a higher timestamp) has already been processed [REED78]. However, this may lead to a "dynamic deadlock" where the same set of transactions is aborted over and over because those

* Garcia [GARC78] demonstrated that for a replicated database a centralized locking scheme performs better than a fully distributed scheme based on timestamps, however, this does not imply that the same will be true for a partitioned database.

transactions repeatedly outdate each other - this is similar to the collision problem in contention networks such as Ethernet [METC76]. A different approach is to discard a newer request in favor of a delayed older request, given that the transaction that generated this newer request has not yet been committed [TAKA78]. This solution may lead to a starvation (i.e., a specific transaction may never succeed since other transactions will always cause it to abort), but it is free of dynamic deadlock. An extension is to allow multiple versions of objects to coexist: a delayed read request can be satisfied without having to abort any other request if the particular version still exists [TAKA78, REED78]. Finally, Montgomery uses an atomic broadcast mechanism that guarantees that all DMs receive their requests in the correct order; that is, "out of order" requests and the associated problems do not occur in this scheme [MONT78].

Phase 2: Commitment and Recovery

The second phase in a DM, actually, the entire time interval between the point when the TM issues the prepare request and when the DM receives the commit or abort request represents a "critical window". It is critical because if the TM fails before it reaches a decision about the fate of the transaction and informs the DMs about this decision, all participants must wait. In some circumstances, this wait period may become unbounded, as it was shown for example by Montgomery [MONT78].

Even if there is no danger that the wait may be indefinite, it may have serious consequences on the performance of the system, since the objects modified by the transaction cannot be used by other transactions until the DMs have received the commit message. The inclusion of another stage, the explicit request to "prepare", is sometimes viewed as a way of shortening the

critical window;* the prepare request could be sent with (be implicit in) the write request.

However, rather than narrowing the critical window, a better approach is to make it less critical. This criticality is a result of assigning the responsibility for making and executing the final decision to a single component. One solution to this problem is to replicate this component. This approach is used by Reed [REED78] where the state of a transaction is maintained in a commit record, a data structure that can be replicated in several nodes. Each DM can determine the state of the transaction by inspecting enough copies of the commit record to obtain a predetermined number of votes. Hammer [HAMM79] also utilizes multiple instances of the coordinator, but the coordinator is replicated using a master/backup arrangement similar to that proposed by Alsberg [ALSB76]. Using this arrangement, the master is responsible for making the decision and delivering the commit (or abort) messages to the DMs, but if it fails, one of the backups will become a new master and complete this phase.

Takagi and Montgomery took a radically different approach. They both rejected the requirement that the objects used in a transaction are not available until the DMs know the outcome of the transaction. Essentially, assuming that each object can be written only once in a single transaction, the new contents can be made available (conditionally) immediately after the write request has been processed.

Not only does this approach make the performance of the system less vulnerable to failures, it can also improve performance in the absence of failures, since it allows a higher degree of concurrency. However, it leads

---

* This approach is used in [LAMP76 and TAKA78]. Gray in his notes calls it a three-phase commit protocol [GRAY78].

to the problem of cascading of backout.* That is, if transaction Ti fails after some other transaction Tj has read (and possibly modified) some objects modified by Ti, both Ti and Tj have to be backed out. Of course, the output of Tj could have been read by Tk, thus having to backout also Tk, etc. However, assuming that failures are rare, this approach may still be preferable, given that mechanisms exist for proper handling of dependent transactions. Two conditions have to be satisfied:

1) during a backout of each individual transaction, the transaction does not need to "reacquire" (in an exclusive mode) any of the objects that it has read or modified in order to undo the changes, and
2) it is possible to remember (or to reconstruct) all information flow among concurrent transactions.

The first condition is necessary in order to avoid a possibly unresolveable deadlock, that is, a deadlock that cannot be resolved by backing out selected transactions, since backout may cause another deadlock. In such a case the only recovery possible might be resetting the entire system into some earlier consistent state. The second condition follows from the discussion above. An interesting insight emerges from Takagi's work: to be able to successfully manage cascading of backout, the recovery schemes ought to be object-oriented. Object-oriented recovery means that all the information needed to restore an object to some previous value is associated with the object (provided by the manager of the object) rather than with the individual transactions. Since concurrent transactions do not know of each other and their dependencies, it is difficult to properly backout a set of

* This is called the domino effect in the work of the Newcastle group [RAND78].

dependent transactions if the recovery data is maintained by individual transactions. It should be pointed out, however, that transaction-oriented recovery works fine if objects modified by a transaction are not released until the end of the second phase.

It is interesting to compare the approaches used by Montgomery and Takagi. Both maintain "multiple uncommitted versions" of objects together with information of how each version depends on other versions (indirectly, both schemes also provide information about how each version depends on (versions of) other objects). A new version of an object is created when the controlling DM receives the new value for the object; this action does not destroy the old value of the object (that is, the old version). A version represents the (possible) state of the object. In addition to having a value, a version has a time attribute that specifies its range of validity. The range of validity of a particular version is the time interval in the history of the object during which the object was in the state represented by that version. A version is only tentative until the transaction that created it is committed. If the transaction fails, the version is simply discarded. If a version is discarded, that part of the object history is erased. Now if another transaction can read an uncommitted version Vx and create its own version Vy of the same or another object such that the value or even the existence of Vy depends on Vx, it is necessary to remember that Vy is dependent on Vx, since if Vx is discarded, Vy must be discarded also.

In Montgomery's scheme, a request to read an object that currently has several uncommitted versions will return a set of all possible values that the outstanding (not yet committed) transactions could produce. This set is called a polyvalue. Thus, when an object is made visible but before the transaction that modified it is completed (committed or aborted) both the old

and the new value (each of which themselves may already be a polyvalue because the outcome of some earlier transaction has not yet been resolved) are presented to the next transaction. If this next transaction needs a precise answer, it will have to wait. If it is sufficient to know that all values possible as of that time are within an acceptable range, the transaction can calculate new values for each polyvalue component, and if the answer is satisfactory, it may even commit. In this scheme, if one transaction is aborted, no other transactions ever have to be backed out; the only thing that has to be done is to throw away some irrelevant information, thus reducing the polyvalue set. In this sense, the scheme is symmetric for the two possible outcomes of a transaction - this pruning has to be done both for the commit and the abort decision. That is, this scheme, unlike Takagi's scheme, does not make any assumption about the probability of success. More important, it allows transactions to be committed before the earlier transactions that modified the same objects have been committed (or aborted).

Takagi's scheme is more conservative. Here a transaction cannot commit until all the transactions on which it depends have committed. If any such earlier transaction is aborted, all dependent transactions must be backed out. Takagi assumes that failures are rare, that is, once a new version is created, it is very likely that it will be committed; put in different words, with high probability it is the right value that the next transaction should see. Thus, a read request returns the value of the newest version.

Interestingly, Reed also employs multiple versions in his scheme, however, his emphasis are on supporting multiple committed versions. An uncommitted version (token) can be read within the same transaction that created it, but not by other transactions. Thus, no cascading of backout occurs. A transaction is backed out merely by deleting its uncommitted

versions of modified objects. The fact that several committed versions are maintained and that the validity of each version is specified in pseudo-time (an artificial, system-wide monotonically increasing measure of system progress) means that it is possible to get consistent snapshots of the past states of the databases. This ability consequently means that although committed transactions cannot be undone individually, it is possible to back out the entire system to a specific past state.

All the schemes that use multiple versions require two kinds of mechanisms:

1) The existing versions of a particular object must be grouped together (for example, by using a descriptor table [TAKA78] or a linked list [REED78]) and managed in such a way that the users see only a single object.
2) It is necessary to be able to determine whether or not a version is still dependent on the outcome of some transaction and which transaction it is.

In Takagi's scheme, these two kinds of mechanisms are merged in the descriptor table and the operations that manipulate the table. The result is a recoverable object that allows a high degree of concurrency. Each recoverable object, in addition to the operations that read and manipulate the state of the object, has two operations defined on it:

commit: commits the changes made by the transaction that invokes the commit operation, (and only those changes),

undo: restores the state of the object to that in which the object was immediately prior to the processing done by the transaction that invokes the undo operation.

The notion of recoverable vs non-recoverable objects is not new; it appears in the work of Anderson, et al. [ANDE76, ANDE78] and Verhofstad [VERH77]. However, Takagi's definition explicitly brings up the issue of concurrency allowed on recoverable and non-recoverable objects. Reed presents two separate mechanisms, one for creating new versions and incorporating them into the object's history, and a second mechanism, called dependent possibility, for keeping track of version dependencies. Although in the original scheme the latter mechanism is used only to control commitment (or backout) of nested transactions,* such a mechanism could also be used to keep track of information flow between otherwise independent concurrent transactions.

Protocols for Replicated Databases

In the beginning of this paper, the problem of a distributed update was divided into two categories. Thus far, the discussion has concentrated on the first one, that is, update of a partitioned database. However, for reliability reasons, the critical components of a system need to be replicated. An example of the need to replicate a critical component was seen in connection with the function of the TM in the two-phase commit protocol for a partitioned database. The state of such a replicated component can be viewed as a replicated database.

As some people involved in development of commercial computer systems believe, the problem of supporting multiple images of an object in a distributed system is more important than the problem of a partitioned database. This observation has two arguments as its basis. First, it is questionable whether the consistency constraints that span node boundaries

* Transactions can be built of smaller transactions, but the versions produced by component transactions cannot be committed until the containing transaction is committed.

must be maintained at all times. It may be sufficient to ensure consistency of daily reports; temporary inconsistencies that arise during the day may have no effect on the correct operation of the system, due to the nature of the application. Reliability is a much more compelling requirement. Second, maintaining a replicated database seems to be a harder problem than the partitioned database case. On the other hand, it is often not clear that multiple images need be maintained at separate nodes; making a single node superreliable (a multiprocessor configuration) and making sure that it is also always accessible from all the interested parties in the network (by providing alternate communication paths) may possibly be a better solution.

However, there is another important reason for supporting multiple images of objects on different nodes, and that is better performance.* Both of these reasons can be merged into a single goal: the purpose of replication is to increase availability.

Now how is the problem of updating a replicated database different from that for a partitioned database? Conceptually, one might start by observing that a replicated database merely presents a particular example of a consistency constraint, so a two-phase commit protocol could be used to coordinate the update of the existing images. But this approach, while it assures mutual consistency of the images, succeeds in missing a principal point of replicating the database; availability of the database even if some nodes are not operating or accessible. The two-phase commit protocols require

* Actually, one of the main reasons behind a partitioned database is also increased performance. Information should be close to where it is most frequently used; this eliminates the delays that would be experienced if the information had to be acquired from a distant node, especially if the distant node were inoperational or inaccessible. It is hoped that updates that require participation of several nodes will be infrequent. However, even an infrequent update that cannot be completed may delay everybody.

simultaneous availability of all the images in order to accomplish an update. Thus, a more sophisticated scheme must be invented. In particular, what is needed is a scheme where only a fraction of the images must be simultaneously available to be able to proceed with an update. Alsberg's n-resiliency protocol, which requires participation of n images, belongs to this category [ALSB76]. If this approach is taken, it is necessary to ensure that once an update is committed, all images eventually will be updated. Thus, the last part of the two-phase commit protocol, that is, informing all participants about the outcome of a transaction, becomes more involved.

The usual two-phase commit protocol guarantees that all DMs see all the transactions that they ought to see. In the n-resiliency protocols, some DMs* may miss one or several transactions. Since in a replicated database the results of each transaction can always be found out from those images which actually saw the transaction, this is not a problem as long as each image can reliably detect if it has missed something. None of the timestamp-based schemes for transaction scheduling described in the previous section is sufficient to deal with this problem. One possibility is to associate globally unique sequence numbers rather than timestamps with transactions and schedule transactions strictly according to their numbers; that is, before an operation requested by transaction k can be processed by a particular image, that image must have seen all of the transactions k-i, i=1, 2...k-1. Unfortunately, such a scheme requires a central component such as an eventcount [REED77] that assigns these numbers to individual transactions. A central component always poses a reliability problem. So, this component also would have to be replicated, and its updating coordinated, using again some

* In this case, DMs are managers of individual iamges.

n-resiliency protocol.* Finally, to prevent inconsistencies in case of network partitioning, n must be greater than or equal to the majority of the existing images. Such a majority-based two-phase commit protocol was developed by Garcia [GARC78B].

Montgomery's scheme [MONT78] deserves special attention since it does not require any modification to work for a replicated database. Also, it can be easily extended to combine partitioning with replication, that is, some (or all) parts of the database can be replicated. First, his hierarchical network guarantees that all messages are eventually delivered and delivered in the correct order. Furthermore, his scheme does not require a majority vote before a replicated object can be updated, because it is impossible to perform the update unless the request to do so can be sent to all the images. This does not mean, however, that the request must be received by all images before other transactions can read the new value; it only means that the mechanics of the network will guarantee that the update request is processed by all images before any future request that starts on the same or a higher level of the hierarchy can be processed. A later transaction that involves only a few low levels of the hierarchy can possibly be executed before the update of the replicated object is completed. Unfortunately, the hierarchical organization, while it produces the right kind of ordering for transactions involving both partitioned and replicated database, does not lead to a naturally robust system. It may also suffer of poor performance if a large percentage of requests must go through many levels of the hierarchy.

---

* Garcia developed a more elegant solution: the function of the sequence number generator is associated with one of the images, and since at least n images know the last sequence number, they provide the necessary backup [GARC78B].

Building Atomic Transactions

The preceding sections surveyed a variety of schemes developed to ensure atomicity of operations in the face of uncertainties that arise in a distributed (decentralized) system. The next natural question to ask is: how does one choose from these different schemes? Do we know enough about the distributed update problem to be able to choose? Are the individual schemes sufficiently complete, or what else has to be done (known) to ensure that operations will indeed behave "atomically" in a real system?

First, it should be realized that each individual scheme for building atomic transaction that has been reviewed in this paper consists of several concepts and mechanisms that are potentially separable. In particular, the individual mechanisms may have more general use than what is implied by the context in which they were defined and could be applied, either directly or with some small modification, in other schemes. Several enhancements of the reviewed schemes through mechanism introduced in other schemes could be envisioned, for example, the combination of multiple committed and multiple uncommitted versions as suggested earlier. However, based on the studies of the various proposed schemes and, in particular, of the assumptions underlying those schemes, it seems that some of the approaches might be overly conservative and unbalanced in their relative emphasis on different classes of problems. An essential step towards making a significant progress in this area is to get a better understanding of the importance, frequency and severity of the specific problems that the individual schemes for atomic updates attempt to solve. This section describes several issues that in my opinion deserve careful thought.

1. Defining Transactions in Terms of Operations on Abstract Objects

Although the authors of some of the schemes for distributed updates define transaction as (partially ordered) set of operations on abstract objects, without exception, the details of their schemes are worked out for the most general, and most primitive case: the permitted operations on involved objects are read and write. Since operations on abstract objects eventually have to map into reads and writes on the storage representation of the basic objects, such general mechanisms may seem appropriate. However, at such a very low level, two simplifications are highly probable:

1) the objects involved in an atomic operation are all on the same physical node, and
2) atomic operations of this type are not interleaved (i.e., they are executed strictly sequentially).

On the other hand, it might be possible to take advantage of the semantics of transactions that operates on abstract objects. For example, rather than performing (atomically) the following sequence of operations:

i. read the value of account X from node Ni,
ii. add $100.00 to this value at node Nj,
iii. write the new value into account X at node Ni,

a single request "increment account X at node Ni, by $100.00" can be sent by node Nj. If the first approach is taken, not only the number of messages that need to be exchanged between Ni and Nj is larger, but if another transaction tried to increment the same account concurrently, the second read would be delayed, pending (at least) completion of the write part. In Takagi's scheme, if the second read request had a lower timestamp than the first one, the increment operation that performed the first read would be aborted. In Reed's scheme, if the second read request had a higher timestamp than the first one

but arrived before the write request of the first increment operation, again that increment operation that performed the first read would be aborted. If the second approach is taken, the two increment operations could actually be performed in any order, that is, an outdated request to increment the account does not need to be aborted and does not abort the later requests. Of course, it is still necessary to synchronize the reads and writes on the storage representation of the account. However, since the individual read and write requests do not involve sending of messages, it is easier to ensure that they are applied in the right sequence. Takagi extended the definition of conflicts and consistent schedules of concurrent transactions as presented in [ESWA76] (and used as a basis in most of the schemes for distributed updates) to a more general case: two transactions T1 and T2 conflict if T1 performs action a1 on object X and T2 action a2 on the same object and a1 and a2 are not permutable [TAKA78]. Two increment operations on a bank account are permutable; thus, no ordering needs to be imposed on transactions with respect to these operations.* If synchronization constraints can be specified in terms of operations on abstract objects,** determining which operations are permutable is a simple extension.

Thus, in a distributed system the requests that are sent in messages to individual participants of a transaction should represent as high level operations as possible, which in turn means that as much work as possible

---

* Two increment operations are permutable with respect to the final result, but not with respect to time. That is, if the state of the system must reflect the exact history of requests, these operations would not be permutable.

** Synchronization of accesses to an abstract object ought to be specified in terms of the operations defined on the object; some research of this nature has been done in connection with the development of languages that support abstractions [LAVE78].

should be carried locally, by the individual participants (Montgomery's model comes close to this one in that the new values of objects can be computed locally by individual data managers [MONT78]). But this view ought to be carried even further. Abstractly, a transaction is an operation on a single (abstract) object. And this abstract object can be composed also of abstract objects. In reality, the transaction translates into reads and writes on basic objects. However, the individual abstract objects should be recoverable separately and independently, not just as a part of the topmost transaction. This leads naturally to a hierarchical nesting of transactions (atomic operations), that is, each use of an abstract object is an atomic operation. This means that the reads and writes on the basic objects ought to be grouped to reflect the nesting of atomic operations. The mechanism developed by Reed [REED78] supports such hierarchical nesting; it provides a truly general support for implementation of atomic operations on any level of abstraction.

Takagi forcefully argues for object-oriented rather than transaction-oriented recovery, that is, associating the recovery data with part of the object. Since the knowledge of the current use of the object is concentrated at the object itself, it is possible to permit concurrent accesses on a finer level than it would be possible otherwise. This is certainly a worthwhile goal. Reed's mechanism provides object-oriented recovery. It does not exploit fully the potential for concurrency since it does not allow transactions to see uncommitted versions unless it is the transaction that created the version, but as discussed earlier, such an extension is possible.

2. Availability: Reliability vs Performance

Earlier it was said that the purpose of replication is to increase availability, where availability means both that:

1) if a particular node fails or becomes inaccessible, another node will assume the responsibility for providing the services that were provided by the failed node, and
2) performance can be improved if a particular service is provided locally or if there is a contention for a specific service, the contenders can be referred to different images of the service.

The first case can be classified as a reliability problem, the second as a performance problem. My thesis is that these two aspects of availability should be clearly understood, since, taken separately, they will lead to different solutions to handling replicated databases. Certainly, for reliability purposes it is not necessary to allow an unconstrained access to all images, that is, allow both read and update requests from multiple concurrent users. On the other hand, if multiple images are used mainly to improve performance, it may be inappropriate to insist on maintaining mutual consistency. An arrangement that guarantees mutual consistency, while intended to increase not just reliability but also performance may actually constrain availability because of the synchronization overhead.

For many applications, it may not be necessary to support multiple images that are always mutually consistent, that is, it may be sufficient to support coexistence of several versions of an object. However, it is still desirable to be able to view the set of existing versions as one logical object, such that it is possible to determine which of two images is a newer version, if a particular image is the current (newest) version, it two images are the same version, etc. The scheme developed by Reed presents such a view [REED78]; in

this scheme, an object is represented by a history of the states it has had since its creation - each change of state results in a creation of a new version. A version is a read-only entity; thus, versions can be freely copied without having to worry about coordination of update activities on these copies. A flexible means for access specification and control for these different versions was developed by Wyleczuk [WYLE79]. This scheme uses time based capabilities that specify access to a particular version (existing or future version) or the most current version (dynamically changing binding).

3. Forward Recovery

Practically all proposed schemes for implementation of atomic operations employ backward recovery as the way to deal with failures (and scheduling anomalies) encountered in the course of executing such an operation. Backward recovery must be supported, at least as a means for dealing with situations where an intended atomic operation cannot be carried out to completion because the initiator decides to abort it. However, reliability is more than preservation of consistency. In particular, it is the assurance that operations will be performed. Thus, it might be desirable to keep the system going in spite of errors and failures; that is, it is desirable to have facilities for forward recovery.

A forward recovery means that rather than aborting an operation and returning the objects used by that operation to the state immediately prior to the beginning of the operation, an attemppt is made to complete the operation by an alternative means. A replicated database offers a potential for forward recovery; however, no sound protocols for completing a transaction interrupted by a failure have yet been developed.

4. Assumptions About the Underlying System

Most work concerned with reliable distributed update concentrates on the problem of synchronization: scheduling operations of concurrent transactions, and coordination of the participants of a transaction such that either all changes are made or none is made. But upon careful examination, most subtle and most difficult problems arise in the implementation of these synchronization and recovery mechanisms.

It is usually assumed that the system provides:

a) atomic stable storage:

- nonvolatile storage that is guaranteed to survive system crashes,
- write operation is atomic, that is, it either writes correctly or if it fails, the state of the object that was to be modified will be the same as it was prior to the invocation of this operation,

b) atomic message delivery:

either a correct message is delivered or no message is delivered

The latter implies existence of communication protocols that are able to detect bad messages, resend lost messages, and discard duplicate messages. This is a sufficiently well understood ara, and it can be assumed that such facilities are available. Building atomic stable storage from ordinary storage devices (i.e. disk) is a non-trivial matter. Basically, it is necessary to choose a reasonable model of possible failures of the actual hardware devices and build an abstract device that will mask these failures [LAMP76]. It is impossible to build a perfect atomic stable storage; there is

always some non-zero probability that under some circumstances this abstract device will fail.

Atomic stable storage is essential to most of the mechanisms needed to assure atomicity of higher level operations, in particular the recovery mechanisms such as the backout/commit cache used by Takagi [TAKA78] or the transaction log described by Gray [GRAY78]. No matter how carefully the synchronization protocols and recovery mechanisms are specified, in a real system, they will work correctly only as long as the assumption of the atomic stable storage can be satisfied. Thus it is important to minimize the dependence of the synchronization and recovery mechanisms on the existence of such a device.

Another problem that deserves careful analysis is the problem of system partitioning. This problem was brought up in the section on protocols for replicated databases. The problem occurs if the set of existing images is divided into two or more sets (partitions) such that only the images in each partition can communicate. In the spirit of trying to provide service as long as possible, one of the partitions may be allowed to continue. To ensure that under completely decentralized control at most one partition will continue, it is required that such partition must contain the majority of the existing images. To ensure that at least one of the images in such a partition is the most recent version of the database, it is necessary to have a majority vote for any update of the replicated database. Thus, the threat of partitioning leads to very complex and costly update and recovery protocols. It seems that this is a problem that ought to be addressed at a lower level, in the design of the communication network.

Other subjects: The problem of duplicate requests the problem of timeouts (to be completed).

REFERENCES

ALSB76 Alsberg, P.A., "A Principle for Resilient Sharing of Distributed Resources," Proc. of the International Conference on Software Engineering, San Francisco, California, October, 1976, pp. 562-570.

ANDE76 Anderson, T., Kerr, R., "Recovery Blocks in Action: A System Supporting High Reliability," Proc. of the International Conference on Software Engineering, San Francisco, California, October 1976, pp. 447-457.

ANDE78 Anderson T., et al., "A Model of Recoverability in Multilevel Systems," IEEE Trans. on Software Engineering, SE-4, 6, November 1978, pp. 486-494.

ESWA76 Eswaran, K.P., et al., "The Notions of Consistency and Predicate Locks in a Database System," Comm. of the ACM, 19, 11, November 1976, pp. 624-633.

GARC78A Garcia-Molina, H., "Performance Comparison of Update Algorithms for Distributed Databases," Stanford University Digital Systems Laboratory, Technical Note No. 143, June 1978.

GARC78B Garcia-Molina, H., "Crash Recovery in the Centralized Locking Algorithm," Computer systems Laboratory, Stanford University, Stanford, California, Progress Report No. 7, November 1978.

GRAY78A Gray, J.N., "Notes on Data Base Operating Systems," Lecture Notes in Computer Science, 60, Springer-Verlag, 1978, pp. 393-481.

GRAY78B Gray, J.N., "Recent Results on a Two-Phase Commit Protocol," (Seminar given at the M.I.T. Laboratory for Computer Science, March 27, 1979), IBM San Jose Research Center, San Jose, California.

HAMM79 Hammer, M.M., Shipman, D "Reliability Mechanisms in SDD-1: A System for Distributed Databases," Computer Corporation of America, Technical Report (in progress).

LAMP76 Lampson, B., et al., "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Research Center, 1976, (to appear in Comm. of the ACM).

LAVE78 Laventhal, M.S., "Synthesis of Synchronization Code for Data Abstractions," M.I.T. Laboratory for Computer Science Technical Report No. 203, June 1978.

MENA78 Menasce, D.A., et al., "A Locking Protocol for Resource Coordination in Distributed Databases," Proc. 1987 ACM-SIGMOD Conference on Management of Data, Austin, Texas, May 1978.

METC76 Metcalfe, R.M., et al., "Ethernet: Distributed Packet Switching for Local Computer Networks," Comm. of the ACM, 19, 7, July 1976.

MONT78 Montgomery, W.A., "Robust Concurrency Control for a Distributed Information System," M.I.T. Laboratory for Computer Science Technical Report No. 207, December 1978.

RAND78 Randell, B., et al., "Reliability Issues in Computing System Design," Computing Surveys, 10, 2, June 1978, pp. 123-165.

REED78 Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," M.I.T. Laboratory for Computer Science, Technical Report No. 205, September, 1978.

STON78 Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," Proc. of Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 1978, pp. 235-258.

SVOB79 Svobodova, L., "Reliability Issues in Distributed Information Processing Systems," IEEE Fault Tolerant Computing Symposium, Madison, Wisconsin, June 1978.

TAKA78 Takagi, A., "Concurrent and Reliable Updates of Distributed Databases," M.I.T. Laboratory for Computer Science, Computer Systems Research Division, Request for Comments No. 167, November 1978.

TRAI78 Traiger, I.L., et al., "Transactions and Consistency in Distributed Database Systems," IBM Research Division, San Jose Laboratory, San Jose, California, 1978.

VERH77 Verhofstad, J.S.M., "On Multi-Level Recovery: An Approach Using Partially Recoverable Interfaces," University of Newcastle Upon Tyne, Newcastle Upon Tyne, England, Technical Report No. 100, May 1977.