MAINTAINING AVAILABILITY OF REPLICATED DATA IN A DYNAMIC FAILURE ENVIRONMENT

Daniel Barbara
Hector Garcia-Molina
Boris Kogan

CS-TR-064-86

December 1986

Maintaining Availability of Replicated Data in a Dynamic Failure Environment †

Daniel Barbara

Universidad Simon Bolivar
Departamento de Matematicas y Ciencia de la Computacion
Caracas, Venezuela

Hector Garcia-Molina
Boris Kogan

Princeton University
Department of Computer Science
Princeton, New Jersey

ABSTRACT

An approach is presented for maintaining high availability in a replicated database system with a failure prone communications network. The status of the network is assumed to change dynamically making the detection of partitions infeasible. The approach is based on restricting the data items transactions can access and on special requirements placed on update propagation.

December 8, 1986

† This work has been supported by NFS Grants DMC-8505194 and DMC-8351616, New Jersey Governor's Commission on Science and Technology Contract 85-990660-6, and grants from DEC, IBM, NCR, and Perkin-Elmer Corporations.

Maintaining Availability of Replicated Data in a Dynamic Failure Environment †

Daniel Barbara

Universidad Simon Bolivar
Departamento de Matematicas y Ciencia de la Computacion
Caracas, Venezuela

Hector Garcia-Molina
Boris Kogan

Princeton University
Department of Computer Science
Princeton, New Jersey

1. Introduction.

A distributed database system becomes partitioned when, due to component failures, there exist several isolated groups whose members cannot communicate with members of any other group. The management of replicated data in a partitioned environment is a difficult task. Transactions running in different groups of the partition could change the data in a way that makes it hard to later integrate values.

In some applications it is undesirable to halt transaction processing whenever a partition occurs. In some cases it is because partitions occur at critical times. For instance, in military applications a conflict can make communications unreliable and it is precisely at this time that one wishes to have access to the data. In other cases, partitions are the normal mode of operation. For example, in the Unix UUCP network computers are usually disconnected from the network. Periodically, they dial other computers and exchange data. In such an environment one would like to process transactions even when there is no communication. In yet other cases, partitions are not as common, but halting transaction processing causes a serious inconvenience or economic loss. For example, an unavailable airline reservations system means lost customers.

Also note that even in conventional networks, partitions may be more common than what one might initially imagine. First, partitions do occur when gateways connecting networks fail

or when Ethernet cables are inoperative (e.g., when an Ethernet terminator is removed, say to add a new segment). Second, in most cases it is impossible to distinguish a node failure from the disconnection of the node from the network (e.g., because its ARPANET IMP or its Ethernet transceiver failed). Hence, every node failure must be considered a partition. Finally, "virtual" partitions occur when a computer does not respond promptly to network messages. That is, all network protocols have a timeout period. If a computer does not respond within this time limit, it is declared failed, even though it could be operational but slow. The timeout period has to be set relatively short, else it would take an unreasonable amount of time to detect and compensate for real failures. This means that overloaded or slow computers may occasionally cause virtual partitions when they respond late. In summary, even though partitions may not be frequent events, they may occur with enough frequency so that halting transaction processing every time they arise is not acceptable.

One approach for coping with partitions, followed by many systems, is to allow processing in at most one of the groups. Voting mechanisms can be used to implement this technique [Giff83]. The integration of values after the repair of the partition amounts to broadcasting the changes to the nodes in other groups. This strategy has the advantage of requiring no knowledge about the transactions that run in the system, and thus is a general approach. It has, however, the drawback of limiting the availability of the data during a partition to the active group in the system. Furthermore, note that in some partitions no group will have a majority of votes, and hence there will be *no* active group.

Another approach, followed by [Davi82] and [Park81] allows multiple groups to continue processing transactions. Protocols are designed to detect, after the partition disappears, the conflicts among the transactions that were run. In some cases, the backing out of a finished transaction is necessary to guarantee the consistency of the overall schedule. Note that serializability may not be preserved. Such strategies are called optimistic since they assume that the number of conflicts will be small.

Recently, D. Skeen and D. Wright [SkWr84] described a technique to increase the availa-

bility in partitioned systems, while avoiding conflicts at the same time. Their approach divides transactions into classes, and assigns classes to groups in the system. Analyzing a class conflict graph, the system can decide whether or not to run a certain class of transactions in a particular group of the partition.

This approach successfully increases concurrency in the partitioned system. However, it forces the system designer to classify the transactions. This restriction may be a drawback in systems of heterogeneous users, where users submit unpredictable transactions. More important yet, the approach assumes that the nodes belonging to a group have complete information about the members of the group, and that the partition borders do not change dynamically. For instance, this implies that all the links whose failure caused the partition are repaired at the same time. We call this type of behavior *clean partitions.* The notion of clean partitions may not represent a realistic scenario. Consider a network where failures of communication links occur fairly often and repairs do not take long. That will result in the rapidly changing status of the communication network. Moreover, as pointed out earlier, significant communication delays are not easily distinguishable from failures. Clearly, under such conditions it will be extremely difficult to detect when partitions occur and when they are repaired. It will also be difficult to determine the exact constituency of every group.

Note that the other two approaches mentioned above suffer from the same limitation. There are known techniques for maintaining availability in a dynamic failure environment, but they usually do not guarantee serializability [Davi85], [Sari85].

In this paper we propose a solution to the management of transactions in a replicated distributed database environment that provides a high degree of availability in the face of communications failures and does not depend on detection of partitions. The technique defines a universal mode of operation for a distributed database system that is to be in effect at all times, regardless of the current status of the communication network. The approach will guarantee global serializability of transaction schedules by restricting transactions' read and write access rights in a certain systematic way and placing special requirements on the propagation of

updates. We will allow nodes of the system to process transactions freely, without any explicit synchronization with other nodes in the system. The resulting scheme has some strengths and weaknesses that we will analyze later in the paper.

The paper is organized as follows. In Section 2, we describe the general basis for our technique. In Section 3, we describe the protocol that handles update propagation. Section 4 will focus on extensions that enhance the approach. Finally, in Section 5 we will draw some conclusions.

2. The Model.

A distributed database system is viewed as a set of data items and a set of nodes interconnected by some communication network. We assume that the database is fully replicated[†], i.e., each node has a complete copy of it.

In our scheme, the entire database is logically divided into k non-overlapping subsets of data items, called *fragments*, denoted F_1, F_2, ..., F_k. Each node has a copy of every fragment. Each fragment F_i is controlled by a unique node $N(F_i)$ responsible for updating items in the fragment. We call the process at $N(F_i)$ that controls F_i the *agent* of F_i, or $A(F_i)$. For simplicity we assume that each node (and agent) controls only one fragment.

Each transaction executes at a single node and can only update the fragment whose agent is local to that node, but it can, in general, read copies of other fragments. (Since there are local copies of all fragments, these reads are performed at the updating node.) Subsequently, all resulting updates are propagated throughout the system. Each node has a local concurrency control mechanism used to schedule transactions within that node. However, there is no (explicit) synchronization among the nodes. Therefore, each node can process transactions independently and regardless of the current status of the communication network. We will also assume that all nodes control some fragment. (Nodes that do not can still be considered as controlling

[†] This assumption is purely for the sake of simplicity. In real systems, complete replication is not common, but the most valuable and frequently accessed data is replicated to improve reliability.

an empty fragment.)

Some of the restrictions above can be easily dropped. For instance, an agent could very well be implemented as a collection of processes running on a group of nodes, as long as these processes follow a global distributed concurrency control algorithm to update the fragment controlled by the agent. Also, the restriction that a transaction only updates one fragment can be eliminated, as we will discuss later, albeit with some loss of availability. However, the notion of having only one agent writing a fragment is fundamental to our technique.

Given the fragments, we are interested in knowing what restrictions have to be imposed on the reading of items outside the fragment in order to guarantee a serializable schedule. To characterize these restrictions, we define the following graph.

Definition 2.1. The *read–access graph* (RAG) is a directed graph $G = (V, E)$, where $V = \{F_1, F_2, \ldots, F_n\}$ and $E = \{(F_i, F_j) : i \neq j$ and there is a transaction T that is initiated by $A(F_i)$ and can read a data object contained in $F_j\}$.

Figure 2.1 shows an example of a read-access graph with three fragments: F_1, F_2 and F_3. Suppose that $A(F_i)$ resides at node i, for $i = 1, 2, 3$. Then transactions that run at node 1 can read items in fragments F_1 and F_2, those running at node 2 can read items in fragments F_2 and F_3, and those running at node 3 can read items in fragments F_1 and F_3.

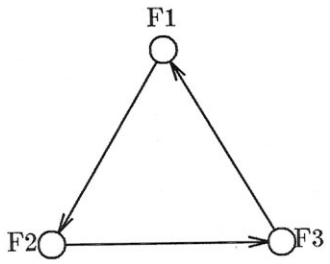


Figure 2.1

The restriction that we want to impose on RAGs is that they be acyclic. To show the need for this, consider the example of Figure 2.1. Suppose that a partition occurs and leaves each node in complete isolation. Our goal is to allow nodes to continue processing their transactions.

However, if the nodes do this, according to the sample *RAG*, the global schedule may not be serializable. For instance, if transaction T_1 runs at node 1, reads $b \in F_2$, and writes $a \in F_1$, transaction T_2 runs at node 2, reading $c \in F_3$ and writing b, and transaction T_3 runs at node 3 reading a and writing c, the results cannot be integrated to match any serial schedule. Since the nodes must be able to operate autonomously to deal with dynamic partitions, acyclicity is fundamental. Therefore, the application must be structured to guarantee that the corresponding graph is acyclic.

The above read and write access restrictions may not be acceptable in some systems. On the other hand, we are convinced that there are many applications simple enough to painlessly incorporate these restrictions. Let us consider an example. Figure 2.2 shows the acyclic *RAG* of an airline database system that handles flight reservations. We have divided the database into three fragments: seat assignments, reservations and flight schedules. Each fragment is managed by an agent located at a separate computer node. (The computer that handles flight schedules may reside at the airline headquarters, the one managing reservations in a travel agency and the one that manages seat assignments at the airport.) The restrictions over the read access rights are quite natural. Transactions updating seat assigments need to read all three fragments, while those that make reservations only need to read the fragment containing flight schedules and their own fragment. Finally, transactions that update schedules do not need to access any other fragment.

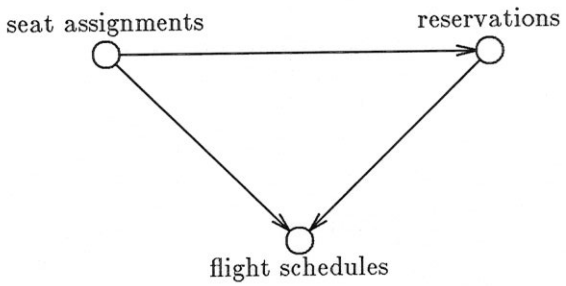


Figure 2.2

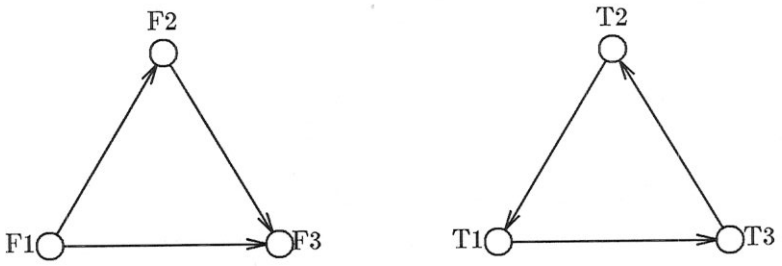
In an actual airline reservation system, we may have more than one fragment for the seat

assignments (one for each airport, for instance) or for the reservations (one for each scheduled flight), but that only will increase the complexity of the graph without creating a cycle. Also, any of the fragments may be managed by an agent that resides in a group of nodes (several travel agencies, each using its own computer, may handle the reservation updates). But, as long as the group of nodes follow a global concurrency control for the transactions, the basic idea will be the same.

3. Update Propagation.

As was stated in Section 1, our goal is to devise a scheme that would provide for a uniform way to operate the database, that is, uniform with regard to the status of the communication network. Such a scheme would do away with the necessity for partition detection that is normally done in traditional approaches in order to know when to switch from a regular distributed concurrency control to a special partitioned mode of operation.

In the absence of any other synchronization provisions, acyclicity of the *RAG* still does not guarantee serializable execution in a dynamic failure environment. To see this consider the example shown in Figure 3.1. The database consists of three fragments: F_1, F_2, and F_3. Let a, b, and c be data items such that $a \in F_1$, $b \in F_2$, and $c \in F_3$. The *RAG* is acyclic, but here is how a non-serializable schedule can arise.



(a) RAG (b) Serialization graph

Figure 3.1

Suppose $A(F_1)$ initiates transaction T_1: $[(T_1, r, c), (T_1, r, b), (T_1, w, a)]$;[†] $A(F_2)$ initiates T_2: $[(T_2, r, c), (T_2, w, b)]$; and $A(F_3)$ initiates T_3: $[(T_3, r, c), (T_3, w, c)]$. Suppose further

[†] A parenthesized triplet denotes an atomic action, with the first element identifying the transaction it is

that update $(T_2,\ w,\ b)$ is installed in the local copy of the node at which $A(F_1)$ is located before action $(T_1,\ r,\ b)$ is executed (generating dependency edge $T_2 \rightarrow T_1$ in the serialization graph); action $(T_1,\ r,\ c)$ is executed before update $(T_3,\ w,\ c)$ is installed at the node of $A(F_1)$ (generating precedence edge $T_1 \rightarrow T_3$); and finally, $(T_3,\ w,\ c)$ is installed at the node of $A(F_2)$ before $(T_2,\ r,\ c)$ is executed (generating dependency edge $T_3 \rightarrow T_2$). The described sequence of events yields a cyclic serialization graph, and hence, a non-serializable schedule.

To avoid such problems, our scheme also restricts the ways in which updates propagate in the network. In Section 3.1 these restrictions are specified , and in Section 3.2 it is shown that acyclic *RAG*s and our update propagation mechanism are sufficient to guarantee serializability of transaction execution.

Before we proceed along the formal track, however, it would be helpfull to study a little further the simple example that we have touched upon (Figure 3.1). No attempts have been made to regulate how and when the updates should be propagated. As a result, a situation occured which lead to the non-serializable schedule. One way to explain what went wrong is to look at the sequence in which updates are installed at the nodes. As it happens, the update generated by transaction T_3 is installed at node 1[†] only after the update of transaction T_2.

On the other hand, these same updates were installed at node 2 in the reverse order. It is easy to see that if the order of installation of updates at node 1 followed that at node 2 (T_2 after T_3), the schedule would be serializable. One way to achieve this is to have node 1 accept update messages originating at node 3 only from node 2 ($3 \rightarrow 2 \rightarrow 1$). In addition to that, node 2 will make sure that if an update from node 3 is installed locally before a local update, then the former will be dispatched to node 1 first.

In the next two sections this idea is generalized, and the resulting protocol is proven to be sufficient to achieve serializability.

---

part of, the second element specifying the type of action (read or write), and the third element identifying the data object on which the action is performed. The entire expression in brackets denotes the ordered sequence of actions comprising the transaction. See [Eswa76] for details.

[†] For brevity, we say, hereafter, "node i" instead of "the node of agent $A(F_i)$."

3.1. The Chain Update Propagation Protocol.

Since we have assumed that there is exactly one agent at every computer node in the system, the nodes of the *RAG* can be viewed alternatively as computer nodes where the agents for respective fragments reside. Let all the nodes be numbered 1 through n, according to some topological order (which must exist if the graph is acyclic), i.e., for any i and j such that $1 \leq i < j \leq n$, there is no path from j to i in the *RAG*. Let $U(T_k)$ denote the list of updates produced by transaction T_k.

Informally, the propagation protocol can be described as follows. All updates flow along the logical chain of nodes $n \rightarrow n-1 \rightarrow \cdots \rightarrow 1$. This means that before an update from node i can be installed at node j ($i > j$), it has to be installed at every node k such that $i > k > j$. Furthermore, upon receipt of a message from node $i+1$ containing a list of updates to be installed, node i procedes with the installation, then puts at the head of the list all the updates generated locally since the last dispatch was sent out to node $i-1$, and sends the enlarged list to $i-1$. The intent of all this is to ensure that all updates are installed in the same order systemwide. We shall see soon why this is important. Note, in conclusion that the above mentioned chain of nodes is *logical* and may not coincide with the physical chain of propagation, i.e., the update messages do not have to go *directly* from node i to node $i-1$. However, after updates are installed at node i they must be installed at node $i-1$ before any other node. It may be possible to route the message from node i to node $i-1$ via some intermediate nodes as long as those nodes simply pass the message along without attempting to install the updates.

We procede now with a more precise specification of the protocol. To avoid a possible confusion the reader should keep in mind that any possible descrepancies between the informal description above, on the one hand, and some of the details that follow, on the other hand, are due solely to the desire to keep the former as simple as possible, and to build up an intuitive understanding.

Every node i will maintain two lists, which are initially empty. They are $UPDATES(i)$ and $OUT(i)$. The protocol consists of three concurrent procedures.

(1) *(Permanently in execution at node i, $1 \leq i \leq n$.)* For every local tranaction T_j, append $U(T_j)$ to $UPDATES(i)$. The order in which different transactions' updates are appended to the list is determined by the local serialization order.

(2) *(Executed at node i, $1 \leq i < n$, upon receipt of $OUT(i+1)$.)* Append $OUT(i+1)$ to $UPDATES(i)$ and atomically install all updates from $OUT(i+1)$ in the local copy of the database. Make sure that for every local transaction T_j which is serialized before these updates (and none other) $U(T_j)$ is appended to $UPDATES(i)$ before $OUT(i+1)$.

(3) *(Iterated with arbitrary intervals at node i, $1 < i \leq n$.)* Atomically copy $UPDATES(i)$ to $OUT(i)$. Send $OUT(i)$ to node $i-1$. Reinitialize $UPDATES(i)$ to empty.

Note: Since $UPDATES(i)$ is accessed concurrently by several procedures, it is essential that a locking mechanism be used.

How often and when procedure (3) is executed has no bearing on the correctness of the protocol. It may, however, affect its efficiency and performance.

3.2. Correctness of the Protocol.

Lemma 3.1. If for every edge (T_i, T_j) in the serialization graph, every update from $U(T_i)$ is (effectively) installed systemwide before any update from $U(T_j)$, then the serialization graph is acyclic.

Proof. Suppose not. Let $C = (T_1, \ldots, T_k, T_1)$ be a cycle in the graph. Then every update from $U(T_1)$ must be installed systemwide before itself, which is an absurdity. □

Lemma 3.2. The protocol of Section 3.1 guarantees that for every edge (T_i, T_j) in the serialization graph, every update from $U(T_i)$ is (effectively) installed systemwide before any update from $U(T_j)$.

Proof. Consider any pair of transactions T_i and T_j such that (T_i, T_j) is in the edge set of the serialization graph.

Case 1. Transactions T_i and T_j are local to the same node, say node q, $1 \leq q \leq n$. Since (T_i, T_j) is an edge in the graph, T_i must be serialized (locally) before T_j. Hence, $U(T_i)$ is

effectively installed in the local copy of the database before $U(T_j)$. Moreover, they are appended to $UPDATES(q)$ in the same order, and thus, will be installed in that order in the copies at nodes $q-1$, ..., 1. Since all other nodes (n, ..., $q+1$) cannot read data written by T_i and T_j, we can say that the same order of installation holds at those nodes as well (even though $U(T_i)$ and $U(T_j)$ never actually reach them).

Case 2. Transaction T_i is local to node q, transaction T_j is local to node r, and $1 \leq q < r \leq n$. This means that T_j overwrites a value read by T_i. Therefore, T_i is serialized at node q before $U(T_j)$. Consequently, $U(T_i)$ is appended to $UPDATES(q)$ before $U(T_j)$. This, in turn, implies that nodes $q-1$, ..., 1 install $U(T_i)$ before $U(T_j)$. Since nodes n, ..., $q+1$ cannot read data written by T_i, we can say, as before, that the same installation order applies at these nodes.

Case 3. As in Case 3, except $r < q$. Here, T_j reads a value written by T_i. Therefore, $U(T_i)$ is serialized at node r before T_j, $U(T_i)$ is appended to $UPDATES(r)$ before $U(T_j)$, and finally, $U(T_i)$ is installed at nodes $r-1$, ..., 1 before $U(T_j)$. Similarly to Case 2, we say that the installation order is the same at nodes n, ..., $r+1$.

Cases 1, 2, and 3 exhaust all possibilities. Thus, $U(T_i)$ is installed systemwide before $U(T_j)$. □

Lemmas 3.1 and 3.2 together establish the following theorem.

Theorem 3.1. The protocol of Section 3.2 guarantees a serializable schedule for any execution characterized by an acyclic *RAG*.

Note that acyclicity of *RAG* was not used explicitly in the proof of the lemmas. This property is essential, however, for without it the logical chain of propagation (of the type required by the protocol) cannot be formed.

3.3. Discussion

Under the protocol we have described, each node operates quite autonomously. Whenever possible it receives updates from the node that follows it in the topological order, and it sends

out updates to the node that preceeds it. However, if these updates cannot be received or sent, the node is *not* blocked. It continues processing its local transactions. The node does not coordinate the execution of its transactions with other nodes (e.g., with a two phase commit protocol), nor does it care if the network is partitioned or not. In this sense, the database is always available for transaction processing.

Of course, we do not get all this for free. As we know, a node is not free to execute any transaction it wishes. It can only run those that do not cause *RAG* cycles. Furthermore, although consistency is preserved, the transactions may operate on data that is not up to date. To illustrate, let us return to the flight reservation example of Figure 2.2. Suppose that the flight schedules node cannot communicate with the reservations node, but all other communications are possible. Say a transaction modifies the schedules. This update will not be received by the reservations node, and it will continue to make reservations based on the "old" schedule. The seat assignments node could receive the latest schedule, but because of the chain propogation protocol, it does not. Hence it also operates with the "old" schedule.

We see then that failures cannot halt transaction processing and cannot cause the database to be inconsistent. However, they can slow down the propagation of updates.

4. Extensions.

The scheme presented in the last two sections can be enhanced in two fundamental ways: by modifying the chain protocol of Section 3.1 so as to speed up the propagation of updates, and by removing some of the restrictions of the scheme (possibly at the cost of decreasing availability).

This section is devoted to exploring these possibilities. However, the reader should view the material assembled here as representing not necessarily a completed work but rather a collection of ideas currently under development.

4.1. A Speedier Propagation of Updates.

In Section 3 we showed that if update propagation is done in a certain controlled fashion, serializability can be achieved without any explicit synchronization. However, the chain protocol incurs the price of a slower propagation of updates, as compared to unrestricted broadcasting of updates. While it was demonstrated that the latter may lead to unserializable behavior, there is nothing to indicate that one cannot have a protocol which gives a speedier update propagation than the "chain" protocol and still enforces serializability.

Once again, let us turn to an example. Figure 4.1 depicts an acyclic *RAG* with the three nodes numbered topologically.

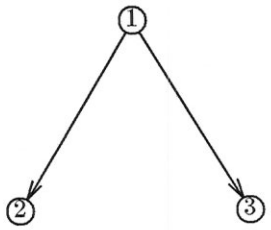


Figure 4.1

The chain protocol would dictate that updates from node 3 have to be installed at node 2 before they can be accepted by node 1. It is not hard to convince oneself, however, that having update messages go directly from 3 to 1 bypassing 2 will never violate serializability. Thus we can make updates from node 3 more readily available at node 1 at no cost in terms of correctness.

This example may suggest that instead of sending updates along the linear topological chain one can propagate along all available paths in the *RAG* (in the reverse direction). Another example, however, will show that this conclusion is a little hasty. Consider the database described by the *RAG* of figure 4.2. Suppose that update messages are sent along two routes: $4 \rightarrow 2 \rightarrow 1$ and $4 \rightarrow 3 \rightarrow 1$. Let a, b, c, and d be data items belonging to fragments F_1, F_2, F_3, and F_4 respectively.

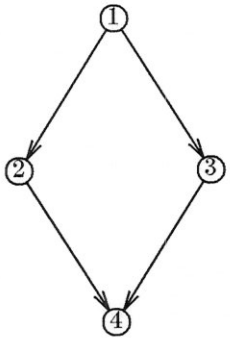


Figure 4.2

Let T_1, T_2, T_3 and T_4 be transactions that execute at nodes 1, 2, 3, and 4 respectively. They are as follows:

T_1: $[(T_1, r, b),(T_1, r, c),(T_1, w, a)]$
T_2: $[(T_2, r, d),(T_2, w, b)]$
T_3: $[(T_3, r, d),(T_3, w, c)]$
T_4: $[(T_4, r, d),(T_4, w, d)]$

Consider the following sequence of events at every node.

At node 4:

(T_4, r, d)
(T_4, w, d)
Updates sent to nodes 2 and 3

At node 3:

Updates from node 4 received and installed
(T_3, r, d)
(T_3, w, c)
Updates sent to node 1

At nodes 2:

(T_2, r, d)
(T_2, w, b)
Updates sent to node 1
Updates from node node 4 received and installed

At node 1:

Updates from node 3 received and installed
(T_1, r, b)
(T_1, r, c)
(T_1, w, a)
Updates from node 2 received and installed

The serialization graph that corresponds to the above schedule is shown in Figure 4.3(a).

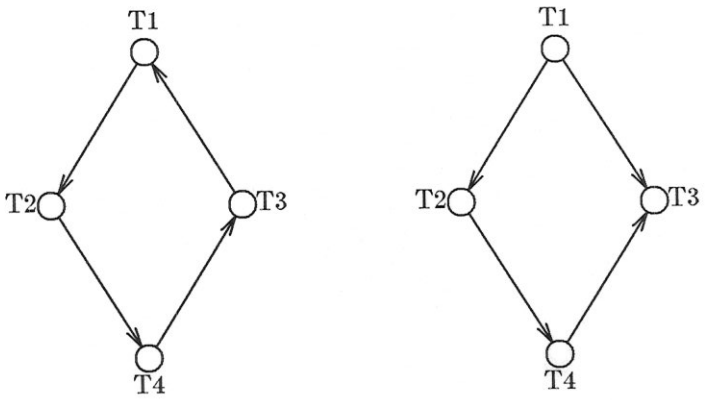


Figure 4.3

It is cyclic, and thus, the schedule is non-serializable. The problem is caused by the fact that transaction T_1 reads inconsistent versions of data items b and c, i.e., the version of b is computed based on the old version of data item d, whereas the version of c is computed based on the new version of d. To guarantee that this cannot happen, we shall require that node 1 install the updates forwarded from node 2 and those from node 3 as an atomic unit. This implies that if one of these messages is received, it will not be installed until the other arrives as well. This rule will change the sequence of events at node 1 as follows.

At node 1:

Updates from node 3 received
$(T_1,\ r,\ b)$
$(T_1,\ r,\ c)$
$(T_1,\ w,\ a)$
Updates from node 2 received
Updates from nodes 2 and 3 installed

The serialization graph corresponding to the new schedule is shown in Figure 4.3(b). One of the edges has been redirected by following the suggested rule, which broke the cycle and rendered the schedule serializable.

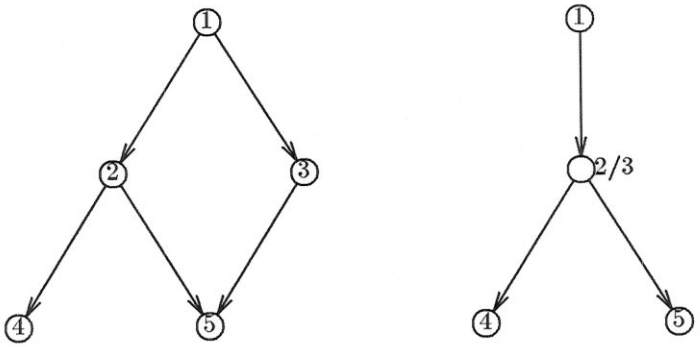
We believe that the rule can be generalized to produce a new update propagation protocol which would have the property of delivering updates faster than the "chain" protocol and would

still guarantee serializability (although a formal proof of the latter is unavailable at present).

4.2. Multiple Agents at a Node.

Until now, we have required that only one agent be at every node in the system. However, there is no reason for this restriction other than for making the exposition simpler.

There are two ways to deal with multiple agents at one node. One way is to replace conceptually the entire group of such agents by one agent. Practically, it means that instead of having several types of transactions, with different access privilleges that could run at the given node, one will have just one type with wider privilleges. To be more specific, every transaction running at the given node will be able to write any of the fragments whose agents are at the node and read any of the fragments that transactions initiated by these agents were allowed to read.



(a) Before (b) After

Figure 4.5. Collapsing two nodes of the RAG into one.

To illustrate consider the *RAG* of Figure 4.5. Suppose that agents 2 and 3 are located at the same computer node (Figure 4.5(a)). Then the corresponding nodes of the *RAG* will be collapsed to form one node as shown in Figure 4.5(b).

Such fusion of nodes, however, may lead to problems. Namely, an acyclic *RAG* may become cyclic as a result of it, as the example in Figure 4.6 indicates.

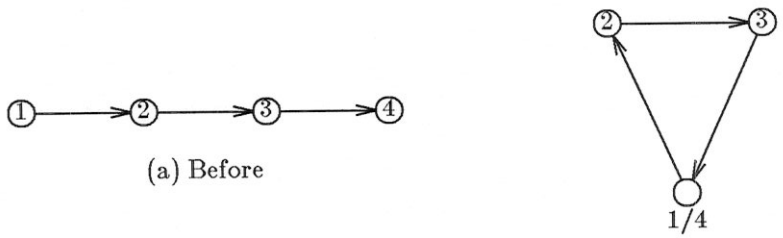


Figure 4.6. Collapsing of nodes introduces a cycle in the RAG.

When this type of situation occurs, fusion cannot be done. Instead, the system should maintain logical separation of agents at the computer node. Transactions initiated by one agent cannot update fragments controlled by another. As far as the update propagation protocol is concerned, it should treat the node with multiple agents as several distinct virtual nodes. Thus, an update sent to the virtual node of one of the agents cannot, in general, be shared with other agents located at the same physical node.

4.3. Transactions that Update Multiple Fragments.

One of the underlying assumptions in our scheme was that every transaction can update at most one fragment — the fragment controlled by the agent initiating the transaction. Although, in Section 2, we argued (with the help of an example) that this restriction is not as limiting as it might seem at first, one has to admit that there well may be circumstances in which it will be necessary to execute a transaction that updates more than one fragment. To that end, the following exception-handling provision can be incorporated into the scheme, albeit at some cost in availability whenever it is used.

To update a fragment which it is normally not allowed to write into, a transaction must obtain exclusive locks on the copy of this fragment (or a part of it) at every node with an agent that can read the fragment. Then the transaction proceeds to install the update atomically at all those nodes. And after that the locks can be released. Note that this mechanism is very similar to a number of commonly used concurrency control algorithms for distributed systems.

The described provision guarantees that the conditions of Lemma 4.1 are satisfied, and thus serializability is preserved. An optimization is possible here. Suppose transaction T_x updates several fragments controlled by agents located at nodes numbered higher than j (in the topological order). Then the updates of T_x should be installed atomically at nodes j, $j+1$, ..., n. However, those same updates can be propagated to nodes 1, ..., $j-1$ in the usual manner.

4.4. Violations of the *RAG* Restriction.

Just as it may be necessary to run transactions that update more than one fragment, it may also be necessary to run transactions that would (or could) create a cycle in the *RAG*. Again, the solution here is to use conventional locking to run these special transactions. By locking all data involved, these transactions could ensure that transactions that used the chain protocol either fully preceded them or followed them in the serialization order. Failures during the execution of these transactions could cause blocking and decrease availability.

5. Conclusions

We have presented an approach for managing replicated data in a dynamic failure environment. It yields a high degree of node autonomy and data availability while ensuring serializability. Partitions do not have to be detected and nodes do not need to have a consistent view of the operational state of the nodes and communication links.

The major drawback of this approach is that the read patterns of transactions must be pre-analyzed and that the resulting *RAG* must be acyclic. As discussed in section 4, these restrictions can be periodically relaxed, but each time this is done locking is introduced and availability suffers.

We believe that this is not a limitation of our particular approach, but an inherent property of dynamic failure environments. One simply cannot run arbitrary transactions in a partitioned network and guarantee serializability. If for a particular application it is necessary to

run arbitrary transactions, then one must either give up serializability or the ability to operate in partitioned networks. If an application demands serializability and operation in a dynamic failure environment, then the transactions must be simple enough so that they can be pre-analyzed. Furthermore, it is necessary to structure the application so that the *RAG* is acyclic.

6. Acknowledgments.

We would like to thank the referees for their useful comments.

7. Bibliography.

[Davi84] Davidson, S.B., "Optimism and Consistency in Partitioned Database Systems," *ACM Trans. Database Syst.*, Vol. 9, Num. 3, September 1984, pp. 456-481.

[Davi85] Davidson, S.B., H. Garcia-Molina, and D. Skeen, "Consistency in Partitioned Networks," *ACM Computing Surveys*, Vol. 17, Num. 3, September 1985, pp. 341-370.

[Eswa76] Eswaran, et. al., "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, Num. 11, November 1976, pp. 624-633.

[GaKo86] Garcia-Molina, H., and B. Kogan, "Achieving High Availability in Distributed Databases," Technical Report CS-TR-043-86, Department of Computer Science, Princeton University, June 1986.

[Giff83] Gifford, D.K., "Weighted Voting for Replicated Data," *Proc. 7th Symp. on Princ. of Database Syst.*, March 1983, pp. 8-15.

[Park81] Parker, R.S., *et. al.*, "Detection of Mutual Inconsistency in Distributed Systems," *Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1981.

[Sari85] Sarin, S.K., "Robust Application Design in Highly Available Distributed Databases," Computer Corporation of America, Technical Report, May 1985.

[SkWr84] Skeen, D., and D.Wright, "Increasing Availability in Partitioned Networks," *Proc. 3rd ACM SIGACT-SIGMOD Symp. on Princ. of Database Systems*, April 1984, pp. 290-299.