ACHIEVING HIGH AVAILABILITY IN DISTRIBUTED DATABASES

Hector Garcia-Molina

Boris Kogan

CS-TR-043-86

June 1986

Achieving High Availability in Distributed Databases

Hector Garcia-Molina
Boris Kogan

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

A new approach is presented for managing distributed database systems in the face of communication failures and network partitions. It offers high availability and, at the same time, guarantees some meaningful correctness properties. The approach is based on the idea of fragments and agents. It does not require prompt and correct detection of partitions and other failures.

This work has been supported by NSF 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.

1. Introduction.

Correctness and availability appear to be conflicting goals of distributed database systems. In particular, traditional concurrency control methods guarantee global serializability of transactions but cannot offer high availability in the face of communication failures and network partitions (for an excellent survey of distributed concurrency control mechanisms see [3]). To improve availability, a number of techniques have been recently proposed. Their key is to replicate data at various sites and then to allow transaction processing at any node (or group of nodes), regardless of whether it (they) can communicate with the rest of the system.

We can intuitively view the trade-off between correctness and availability as a linear spectrum of possible solutions (see Figure 1.1). At one end we have global serializability, which is normally

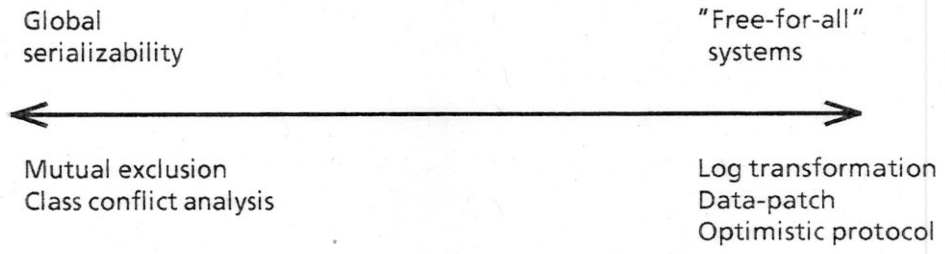


Figure 1.1

considered the correctness criterion for distributed database systems. At the other end is the highest possible availability. We call systems at this end "free-for-all" since they place no limitations on data access during network partitions. From left to right, availability increases while the correctness criteria become less strict.

Previously proposed solutions seem to cluster around either endpoint of the spectrum. (For a detailed survey see [5].) At the left end of the spectrum, we find techniques that guarantee global serializability while introducing some modest amounts of availability for partitioned operations. Examples of such techniques are mutual exclusion ([8]) and class conflict analysis ([10]). At the right end of the spectrum, there are methods that provide practically unlimited availability during partitions at the expense of abandoning global serializability as a correctness criterion. These include the log transformation technique ([2]), Data-patch ([6]), and the optimistic protocol ([4]).

To illustrate the principal differences between the methods from the two extremes we compare the mutual exclusion approach with the log transformation technique as they would apply to a simple hypothetical banking database. Suppose that the information on the balances of funds in different accounts is represented by a table, as in Figure 1.2. The basic types of transactions can be described as follows:

Deposit (Acct#, $) ; *Withdraw (Acct#, $)*

Each transaction specifies the account number and the amount of money to be deposited (withdrawn). Suppose, further, that we have two geographically separated sites, A and B, which are connected by a communication link, and the table is replicated at each of the two (Figure 1.2).

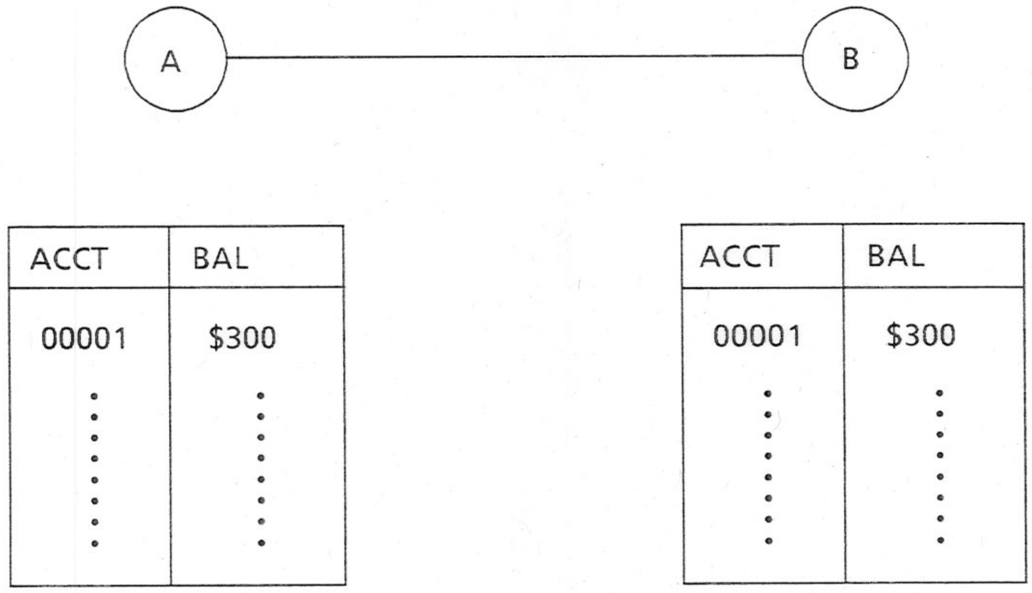


Figure 1.2

Let us examine the behavior of this system when the communication link between A and B has been severed. We will consider two scenarios. In the first scenario,two customers -- one at node A and the other at node B -- make identical requests: *Withdraw (00001, $100)* (we assume that both of the customers have the right to withdraw money from account 00001; in fact, this can be even the same customer, making two withdrawal at different locations). Normally, both of these requests should be granted, for there is enough money in the account to insure a non-negative balance after the withdrawals. But since A and B cannot talk to each other, things get more complicated. Under mutual exclusion, only one of the nodes, say A, can access and modify the data. Therefore, the customer at node A will be able to withdraw his $100; the customer at node B, however, will go home empty-handed. Under log transformation, both nodes are allowed to process transactions, hence both customers will be given the money. However, the correct balance will be established only after the communication is restored. When the nodes are reconnected, they exchange logs for transactions executed during the partition; it is established that the execution happened to be consistent (the balance remained positive), and therefore no corrective action is necessary.

The second scenario is just like the first one, except the amount of money requested by each customer is $200 each. As before, under mutual exclusion, the system will satisfy the request of one customer but not the other. Under log transformation, both transactions will be processed, because neither of them requires the withdrawal of an amount of money exceeding the balance. However, after communications are restored, it will be discovered that the execution was inconsistent, and as a result, the balance went negative. Let us assume the bank's policy is to require that the customer make a deposit to render the balance non-negative and also to charge a fine whenever the account is overdrawn. Then a letter of notification can be sent to the customer and the amount of the fine can be subtracted from the balance.

We have seen that, in the first scenario, the more conservative technique (mutual exclusion) resulted in the loss of service availability, while the more optimistic one (log transformation) insured that both nodes remained operational. In the second scenario, however, the former prevented inconsistent transaction execution, but the latter allowed an account to be overdrawn. So here we have the trade-off between availability and correctness in a very tangible form.

Conservative techniques are quite satisfactory for systems where high availability is not of primary concern. If availability is critical, then "free-for-all" methods seem to make more sense. Each of the variety of "free-for-all" methods has its advantages as well as its shortcomings, but it appears that there are some problems common to all of them. One of these is the computation and communication overhead, which is usually significant and bound to degrade the overall performance of the system. In the above example, sites A and B had to exchange their transaction logs after the partition was repaired. Each of them had to to determine which of the transactions from the received log had to be executed locally and which of the transactions from the local log had to be backed out. More disconcerting than the overhead is the absence of any meaningful, general correctness criteria (aside from eventual convergence of replicated copies) to replace serializability. In other words, it is not clear what properties of transaction execution "free-for-all" systems provide. This may not be a serious drawback for relatively simple (from a semantic point of view) applications such as our example. However, it is increasingly difficult to analyze the behavior of more complex systems without having a formal correctness criterion. In particular, it may be rather difficult to generalize the notion of corrective actions. Note that even in our simple example there are some unresolved issues as far as corrective actions are concerned. Suppose, for instance, that the amount of the fine for an overdrawn account depends on the time that the account in question remained overdrawn. The two nodes can very well have different views of how long the balance stayed negative, and therefore, can impose different fines. The situation can get even more unpleasant if the fine also depends on the actual amount of the overdraft. Specifically, if the nodes charge fines of different amount because of disagreements on the duration of non-negative balances, they can end up with different views of the balance itself. This, in turn, can lead to another round of assessing different fines, and chaos ensues.

In this paper, we concern ourselves with applications requiring high data availability. The goal is to propose an approach that (1) provides high availability in the face of communication delays and network partitions; (2) does not depend on the ability of the system to promptly and correctly detect partitions; and (3) guarantees some useful, formal properties. From the outset, we would like to point out that our approach is not just a new method for dealing with network partitions or a new robust concurrency control mechanism but also a design philosophy for distributed database systems which makes a strong emphasis on high availability.

2. The basic idea and illustrations.

In this section, we introduce the core idea of the proposed approach and give examples to illustrate it. A more detailed and formal discussion is delayed until the next section.

Even though we aim at high availability, in our approach a departure is taken from the "free-for-all" concept which allows any data item to be updated anywhere in the system at any time. The entire body of data is divided into *fragments*, and to every fragment we assign an *agent* (a user or a node with the exclusive privilege to update it). Thus, the decision to update data item x can be taken at node N only if either N itself is the agent of the fragment to which x belongs or the agent is currently at node N. Such a restriction is motivated by the observation that, in real distributed systems, democracy is not a very popular concept, *i.e.*, different users are endowed with different sets of privileges as far as data access is concerned, and the ability of some users to modify data is often restricted. For example, in an airline reservations system, it can hardly be considered a loss of availability if a customer is unable to update the flight schedules in the database. By introducing the notion of fragments and agents we hope to control the way data are updated without sacrificing availability for those who need it.

In this new framework, the banking example from the previous section could be rendered as follows. The table containing the balance information will be unchanged. It will constitute a separate fragment, called BALANCES, with the central office of the bank as its agent (Figure 2.1). In addition,

Balance information

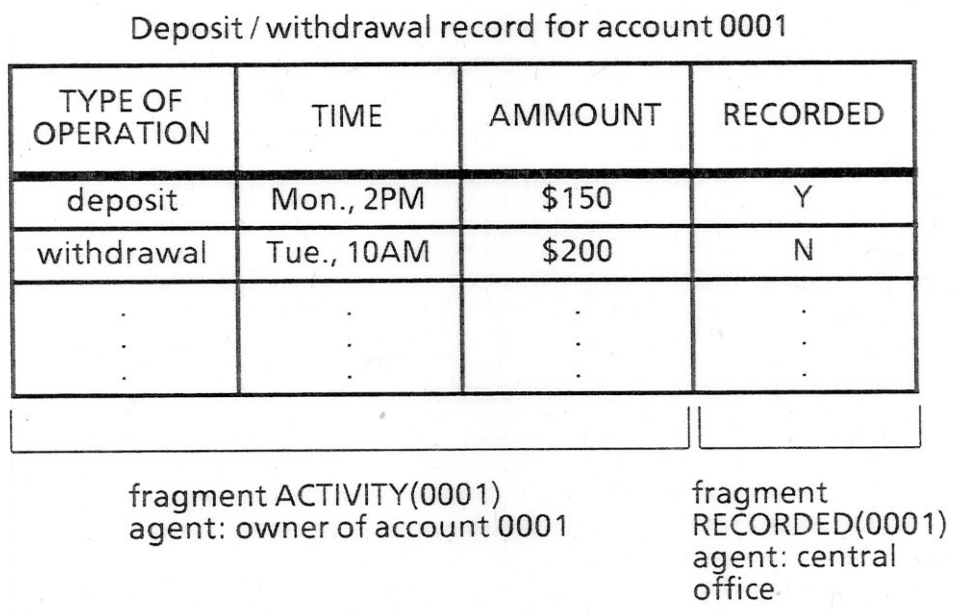
ACCOUNT	BAL
0001	$300
. . .	. . .

fragment BALANCES
agent: central office

Figure 2.1

for each account i, the database will have a table representing the deposit or withdrawal records of that account (Figure 2.2). The entries specifying the type of operation, the time it took place, and the amount of money involved constitute the fragment controlled by the customer (customers) who owns (own) account i. Let us call this fragment ACTIVITY(i). The entry specifying whether the given operation has been reflected in the BALANCES fragment constitutes a separate fragment (call it RECORDED(i)) for which the agent is the central office. Thus, there are two fragments for each account in the bank plus one more fragment for all accounts. Note that the central office is an agent for more than one fragment. For simplicity, we assume that all data are replicated at every node.

At a node other than that of the central office, the local view of the balance (the best guess as to what the real balance is) is computed as:



Deposit / withdrawal record for account 0001

TYPE OF OPERATION	TIME	AMMOUNT	RECORDED
deposit	Mon., 2PM	$150	Y
withdrawal	Tue., 10AM	$200	N
. . .	. . .	. . .	. . .

Figure 2.2

$$\textit{local view of balance} = \textit{balance} + \sum \textit{unrecorded deposits} - \sum \textit{unrecorded withdrawals}$$

where *balance* is the value of the balance field of this account's entry in the BALANCES fragment in the local copy of the database, and *unrecorded* refers to the status of the operation as shown in the local copy of RECORDED. Clearly, in the face of communication delays and partitions, the local view of balance may not correspond exactly to the actual balance. The longer a partition lasts, the greater this discrepancy can become.

The basic banking operations (withdrawals and deposits) can be processed by any operational node, regardless of the status of the communication network. Each such transaction causes an appropriate entry to be made in the ACTIVITY fragment of the account concerned, in the local copy of the database. (The default value for the corresponding RECORDED field will be "N.") Then, this update to ACTIVITY(i) is propagated throughout the network by a broadcast mechanism (if the network is currently partitioned, the propagation will be completed after the partition is fixed) and eventually reaches the central office node. After the update is installed in the local copy of ACTIVITY(i) at this node, a new transaction is triggered here that changes the balance value for the given account in the BALANCES fragment, as appropriate. The same transaction changes fragment RECORDED(i) to reflect the fact that the operation has been seen at the central office. The updates resulting from the transaction are then broadcast to bring the copies of the database at other nodes up to date. Note that updates to a fragment originate only from its agent, as we discussed.

To better understand this example, consider one of the scenarios discussed in Section 1, where two withdrawals of \$200 each are requested (one at every node) during a partitioned operation. They are both granted since the balance is \$300. Let node A be the node that services the central office of the bank. Then the withdrawal that was processed at A will be immediately reflected in the BALANCES fragment. The withdrawal request entered at node B, however, will not reach A until the partition is repaired. When it does reach A, the BALANCES fragment will be updated again, and A will discover that the balance was overdrawn. An appropriate penalty will be assessed (and the resulting update will be communicated to B) through another update to BALANCES, and a letter will be sent to the customer concerned. Note that these actions need be taken only at A, for that is where the agent for the balance information fragment is, and only this agent is allowed to make changes to the fragment. Thus, the decision process involving corrective actions is centralized, and no quagmire, of the sort seen in the example of Section 1, results.

It is worth reemphasizing that the customers are the agents for their deposit / withdrawal records and as such can freely enter requests for bank operations at any node and regardless of the communication status of the network. This is how availability is achieved here.

Note also that a good database design is essential. The example works because the data has been appropriately partitioned according to who should control it. Hence, as stated earlier, we are proposing both a mechanism for accessing the database and a design methodology for achieving controlled high availability.

3. The model.

In this section, a detailed description is presented of our model for distributed databases. The issues of data organization, transaction management, and communications among different sites are discussed, with special attention given to the notion of data control.

3.1. Fragments and agents.

The distributed system under consideration consists of n computer sites, or nodes, interconnected by a point-to-point communication network of arbitrary topology. The database is a set of data objects each of which is replicated at a number of sites. For simplicity, we shall assume from now on (unless otherwise specified) that replication is complete, *i.e.*, every object is replicated at exactly n sites.

External to the system are the users who manipulate the information in the database by issuing transactions. We assume that a user can be connected to at most one node at a time.

The entire database is logically divided into k non-overlapping subsets called *fragments* and denoted $F_1, F_2, ..., F_k$. For future notational convenience, we shall view fragments as sets of data objects. Thus, notation $x \in F_i$ means that data object x is contained in fragment F_i. To control access to fragments *tokens* are used. These tokens, however, are different from the traditional tokens used in distributed systems (see [9], for instance). For every fragment, there is exactly one token, and it can be owned by a user as well as by a computer node. Thus our tokens have existence outside of the computer system and can be passed by means other than electronic messages, a situation quite different from the traditional use of tokens. An update to a fragment can be authorized only by the current owner of the corresponding token, referred to as this fragment's *agent*. That means a node can issue an update to a fragment only if it is itself the agent of this fragment or it has received a request to do so from a user who is the agent. It is not necessary, though, to hold the token in order to be able to read from a fragment, *i.e.*, read actions can be performed freely by all users and nodes on all fragments. This is another difference with traditional token systems.

As an example of a token, consider the card that a bank customer uses to identify himself to an automatic teller. Whoever owns the card is authorized to perform banking operations on the corresponding account, *i.e.*, to update the fragment containing deposit / withdrawal information for that account. One should not infer, however, from this example that all tokens must have a concrete physical embodiment.

Let $A(F_i)$ denote the agent of fragment F_i. We say that N_j is the *home node* of $A(F_i)$, if either $A(F_i)$ is a user and has last issued an update transaction at N_j or $A(F_i)$ is N_j. This semantic twist is but a mere convenience in a context where it is irrelevant whether a particular agent is a node or a user.

2.2. Transaction management and intersite communication.

We distinguish between two kinds of transactions: *update* and *read-only* transactions. It is important to make this distinction, as they are treated quite differently in our model.

Each transaction must be initiated by a unique agent. Let a transaction T be initiated by $A(F_i)$ at the time when $A(F_i)$'s home node is N_j. Then N_j is also said to be the home node of T. We say that T is local to N_j. To all other nodes, T is non-local. Read-only transactions can be initiated by any agent. However, update transactions must satisfy the following requirement.

Initiation requirement: An update transaction T can be initiated by an agent $A(F_i)$ if and only if all data objects modified by T are contained in the fragment F_i.*

When an update transaction is initiated no other node becomes aware of it until the execution completes successfully at the transaction's home node. The home node is then responsible for propagating the updates throughout the system. This is accomplished in the following way. A message is broadcast by the home node, of the form: $(T;\ d_1,\ v_1;\ d_2,\ v_2;\ \ldots;\ d_s,\ v_s)$, where T is the transaction's identifier, d_i is the name of a data object updated by T, and v_i is the new value for that object. Upon receiving such message, every node installs the updates listed in it in its own copy of the database. Here, we wish to emphasize a rather important point: in order to ensure that all copies of the database experience the effects of each transaction, the system does not actually rerun a transaction at other nodes, instead a series of unconditional updates are executed (as explained above), reflecting the desired effects. It is convenient to view a series of such updates spun off from one non-local transaction as a new "write-only" transaction, local to the receiving node. From now on, we shall call these groups of updates *quasi-transactions*. Read-only transactions, naturally, require no propagation.

To implement the necessary message exchanges the system will require a reliable broadcast mechanism which guarantees that (1) all messages are eventually delivered; (2) messages broadcast by one of the nodes are processed at all other nodes in the same order as they were sent.

At every node in the system, a *local* concurrency control mechanism is implemented. As far as this mechanism is concerned, local activities are comprised of update and read-only transactions initiated locally as well as quasi-transactions received from remote nodes. There is a special requirement on the resulting schedule that concerns quasi-transactions: the serial schedule equivalent to it must contain quasi-transactions from a given node in the exact same order as they were generated. The fulfillment of this requirement is important for mutual consistency of replicated copies.

----

*It may appear that the initiation requirement precludes altogether the use of transactions that update more than one fragment. There are ways, however, to circumvent this restriction. One way is to replace, whenever possible, a multi-fragment transaction by a group of transactions that perform the same task and update only one fragment each. When this cannot be done, a semblance of the two-phase commit protocol can be used, that involves the agents of all the fragments that are being updated. For simplicity we only consider single-fragment transactions in this paper. The multi-fragment case is discussed in [7].

4. Control Options.

The notion of fragments and agents we have outlined can lead not just to one but to several control strategies. Each option is characterized by the degree of availability it offers and the type of correctness properties it enforces. The differences among the strategies have to do with how reads are performed and how the movement of agents is controlled. In this section we discuss some of these strategies, and the availability they offer. (Other variations are possible but not discussed here.)

4.1. Fixed agents; read locks.

The most conservative option available fixes all agents at their corresponding home nodes (not allowing them to move) and requires that remote locks be obtained on all data objects that a transaction intends to read (outside the fragment that it updates, if any). For each data object, it is clearly sufficient to acquire the lock on it from the home node of the agent in charge of the fragment containing that object, for that is the only node at which the object can be updated. This option is evidently very close to a number of traditional concurrency control mechanisms and, accordingly can be placed at the global serializability end of our spectrum.

4.2. Fixed agents; acyclic read access pattern.

As before, movement of agents is disallowed, but read locks are no longer required. However, there are certain restrictions on the read access patterns of transactions executing in the system. To characterize these we need to define the following graph formalism.

Definition. The *read-access graph* 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 such that it is initiated by $A(F_i)$ and reads a data object contained in $F_j\}$.

Definition. A directed graph G is said to be *elementarily acyclic* if G_u is acyclic, where G_u is the undirected graph that has the same sets of nodes and edges as G.

Theorem. The transaction execution schedule is globally serializable if the corresponding read-access graph is elementarily acyclic.

Proof: See Appendix.

The above theorem suggests a possible strategy that constitutes a significant improvement in terms of availability over the previous method (now no synchronization is required for reading a fragment) and still insures an important correctness property -- global serializability. To preserve global serializability, however, we had to impose restrictions on which data different transactions can read. This of course poses the question: How reasonable is it, if at all, to demand that these restrictions be incorporated in the database design? There is no unqualified answer. From a general point of view, it is clearly undesirable to restrict the read capabilities of the database transactions. On the other hand, there may be applications for which these demands will not be detrimental in the least. This, of course, calls for an example. The reader should keep in mind that the example that follows is necessarily oversimplified, but it does illustrate our point.

Consider a hypothetical application which keeps track of sales and inventory stock for a wholesale company. There are k warehouse locations at which the merchandise is sold to the customers (retail shops, for example). For every location, there is a fragment in the database that contains a record of every sale made, a record of every new merchandise shipment received at that location, and the quantity on hand of each product. Let us call these fragments W_1, W_2, ..., W_k. Further, there is a fragment C controlled by the company's central office. In this fragment, information will be recorded which represents decisions concerning future purchases (from the manufacturer). These decisions are arrived at by periodic scanning of the contents of fragments W_i and doing the necessary computations. This database is characterized by the read-access graph in Figure 4.2.1. Note that a high degree of

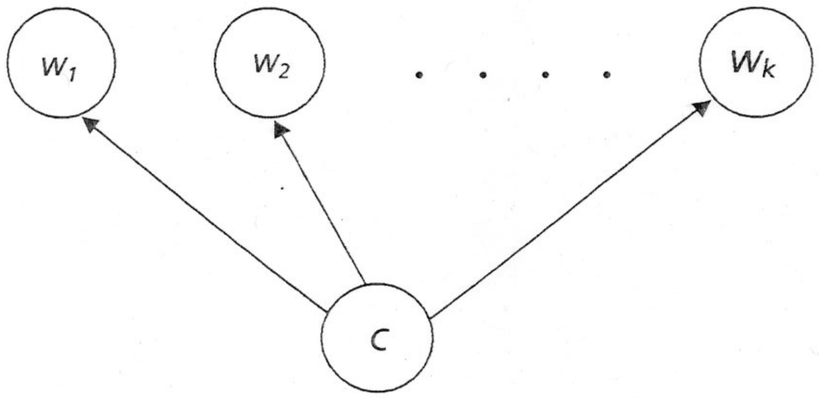


Figure 4.2.1.

availability is maintained by the database. For instance, warehouses enter the sales and shipments information even if there is a communication failure. On the other hand, global serializability is never violated during partitioned operation. In this example, this means that the central site always gets a consistent view of the database.

In certain situations, read-only transactions that violate the restrictions of the read-access graph can be allowed. This is motivated by the fact that potentially non-serializable execution that may result will manifest itself only in the output of these transactions and will not leave any trace on the database itself. If the application at hand is not particularly sensitive to this kind of phenomena, then such transactions can be allowed. In the example above, for instance, one warehouse can be allowed to read from the fragment controlled by another warehouse with no great harm (this can be useful when the current inventory at this warehouse is not sufficient to satisfy a customer's request, and it is desirable to check whether there is more at some other location). In summary, with fixed agents and an elementarily acyclic read-access graph, certain applications may achieve higher availability *and* global serializability. If the read-access graph is elementarily cyclic, it may still be possible to find a subset of transactions that have an elementarily acyclic graph. These transactions, hopefully the most frequent ones, could be executed without read locks, while the rest would be executed with a more restrictive fragment locking policy.

4.3. Fixed agents; no read access restrictions.

Removal of all read access restrictions obviously increases data availability still further. However, the price for that is a possible loss of global serializability. When the read-access graph is not elementarily acyclic, non-serializable schedules may result. Suppose the database consists of three fragments: F_1, F_2, F_3. Suppose further that transactions initiated by $A(F_1)$ read from F_1, F_2, and F_3; $A(F_2)$, from F_2 and F_3; and $A(F_3)$, from F_3 only. The corresponding read-access graph is shown in Figure 4.3.1. It is acyclic, but not elementarily acyclic. Let a, b, and c be data objects, and let $a \in F_1$, $b \in F_2$, $c \in F_3$.

Here is how a non-serializable schedule can arise. 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 that update (T_2, w, b) reaches the home node of $A(F_1)$ and is installed in its local copy before action (T_1, r, b) is executed (generating dependency $T_2 \rightarrow T_1$); action (T_1, r, c) is executed before update (T_3, w, c) is installed in the copy at the home node of $A(F_1)$ (generating $T_1 \rightarrow$

----

*A parenthesized triplet denotes an atomic action, with the first element identifying the transaction it is 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 preformed. The entire expression in brackets denotes the ordered sequence of actions comprising the transaction.

T_3); and finally, (T_3, w, c) is installed at the home node of $A(F_2)$ before (T_2, r, c) is executed ($T_3 \rightarrow T_2$). The above sequence of events yields a cyclic global serialization graph (see Appendix for the definition) as shown in Figure 4.3.2 and, therefore, a non-serializable schedule.

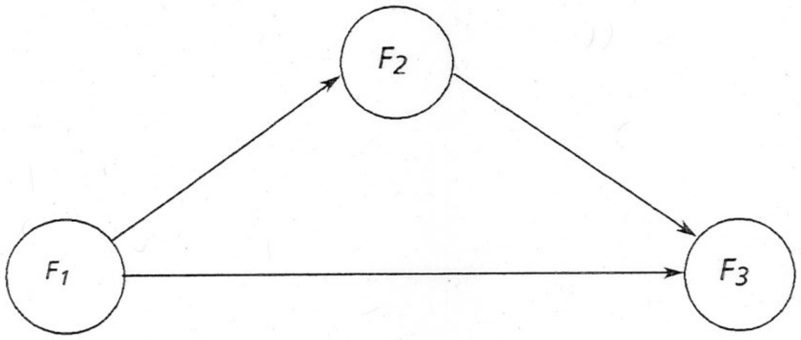


Figure 4.3.1. Read-access graph.

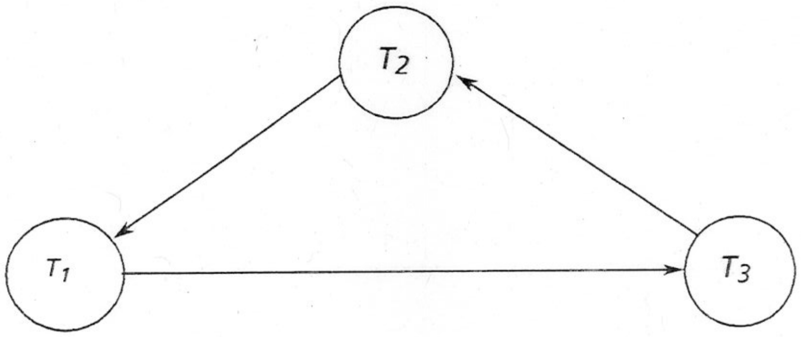


Figure 4.3.2. Global serialization graph.

Short of global serializability, there are some nice properties that are guaranteed by the option being discussed. Let us consider a fragment F_i whose agent's home node is N_j. Since all updates to F_i originate at N_j and are installed eventually in all remote copies of F_i in the same order (see Section 2.2), if at time t the processing of new transactions is halted, and it takes time Δt for all updates to propagate throughout the network, all copies of fragment F_i will be identical at time $t + \Delta t$. The same is, of course, true for every fragment. Thus mutual consistency of all replicated copies of the entire database is guaranteed.

Fortunately, there is more. Let us concentrate our attention on fragment F_i. From the entire set of transactions that execute in the system, let us extract only those that update the contents of fragment F_i and ignore the rest. We denote this subset of transactions $U(F_i)$. Thus, a transaction $T \in U(F_i)$ if and only if T updates F_i. Then we have the following:

Property 1. The schedule consisting solely of transactions in $U(F_i)$, for any i, is (globally) serializable.

The correctness of this property follows from the fact that all transactions in $U(F_i)$ are initiated by agent $A(F_i)$ and, hence, executed originally at the same node. Therefore, they are subject to the same local concurrency control mechanism, which insures the serializability of their schedule. Transactions in $U(F_i)$ give birth to corresponding quasi-transactions which are dispatched to other nodes in the network for update propagation purposes. Upon its arrival to a remote site, each quasi-transaction is submitted to the concurrency control mechanism of that site, whereby the effectual atomic execution of it is achieved. This yields

Property 2. No transaction that reads the contents of F_i, for any i, ever sees a partial effect of a transaction in $U(F_i)$.

Definition. A transaction schedule with properties 1 and 2 is said to be *fragmentwise serializable*.

To analyze the import of fragmentwise serializability, let us examine fragment F_i and how it relates to other fragments and the entire database. Fragment F_i enjoys -- as do other fragments -- a certain degree of autonomy within the database. This autonomy is manifested on two levels: structural and procedural. On the structural level, fragment F_i represents a semantic unit of information (such as an airline company flight schedules or information on the inventory stock at a local warehouse of a wholesale distributor). On the procedural level, modifications of its content are handled exclusively by the designated agent. Consequently, fragment F_i appears to be a more or less self-contained collection of data.

The above notion of autonomy serves as justification for the following conceptual view. Instead of one "large" database divided into several fragments, we have a somewhat different arrangement, namely a group of separate, "small" databases with ongoing communications among them. Update transactions operate on a single small database (a transaction updating fragment F_i is said to be a transaction on database F_i). A read-only transaction that accesses multiple fragments is conceptually replaced by several transactions each of which accesses a corresponding "small" database. As far as a particular database (fragment) F_i is concerned, all other databases (fragments) constitute the "outside world." Thus, when a transaction updating F_i reads from another fragment, the value (values) read can be interpreted as input to this transaction from the outside world. Fragmentwise serializability for the "large" database thus translates into serializability for each constituent database.

To understand the precise nature of inconsistencies that may arise as a result of replacing global serializability with fragmentwise serializability, we turn to the notion of consistency predicates. A predicate $P(v(x_1), \ldots, v(x_r))$, where x_i, $1 \leq i \leq r$, is a data object and $v(x_i)$ is the value of x_i, is said to be a *single-fragment* predicate if $x_i \in F_j$, for some j and all $i = 1, r$; it is a *multi-fragment* predicate otherwise, *i.e.*, a single-fragment predicate spans just one fragment, so to speak, whereas a multi-fragment predicate spans more than one. When global serializability is enforced, consistency predicates are never violated. Fragmentwise serializability does not guarantee, in general, that all consistency predicates hold. However, it is an immediate consequence of this correctness criterion that single-fragment predicates are never violated. Thus the only kind of data inconsistency one can encounter is that characterized by violation of multi-fragment predicates.

To illustrate the difference between global serializability and fragmentwise serializability we give a simple example of a schedule that is fragmentwise serializable but not globally serializable. Consider an airline reservations database and suppose that there are only two flights offered by the airline company and just two customers to service. There is one fragment for each customer and one fragment for each flight. Let us call the fragments controlled by customers C_1 and C_2; the flight fragments will be called F_1 and F_2. Further, suppose that the agents for all four fragments are at different nodes. C_1 and C_2 contain two data objects each: $C_1 = \{c_{1,1}, c_{1,2}\}$, $C_2 = \{c_{2,1}, c_{2,2}\}$. All $c_{i,j}$ are initially zero; later on they can be set by the customers to a positive integer value reflecting the number of seats that the i-th customer *would like* to reserve on flight j. For simplicity, suppose that once set they cannot be reset to a different value, *i.e.*, a customer cannot change his mind. Similarly, F_1 and F_2 also contain two data items each: $F_1 = \{f_{1,1}, f_{2,1}\}$, $F_2 = \{f_{1,2}, f_{2,2}\}$. Every $f_{i,j}$ denotes the number of seats *actually* reserved for the i-th customer on flight j. $A(F_1)$ and $A(F_2)$ periodically run transactions that scan C_1 and C_2. Whenever a new nonzero $c_{i,j}$ is discovered, $f_{i,j}$ is set to its value (unless a potential overbooking is detected). The motivation for having $c_{i,j}$ in this database, in addition to $f_{i,j}$, is to allow the customers to enter their requests for reservations any time they want to, regardless of the current status of the communication network, and, at the same time, to ensure that overbooking does not occur. If data items $c_{i,j}$ did not exist, the system would have to either curtail availability during partitions (suspend accepting reservations at least at some nodes) or allow overbooking to occur. But since the process of making reservation requests is decoupled from the process of deciding which requests are granted, and the latter is centralized (in the sense of being done by just one agent), we get the best of both worlds: availability and correctness.

Figure 4.3.3 shows the read-access graph for this database. Let us consider a schedule in which every

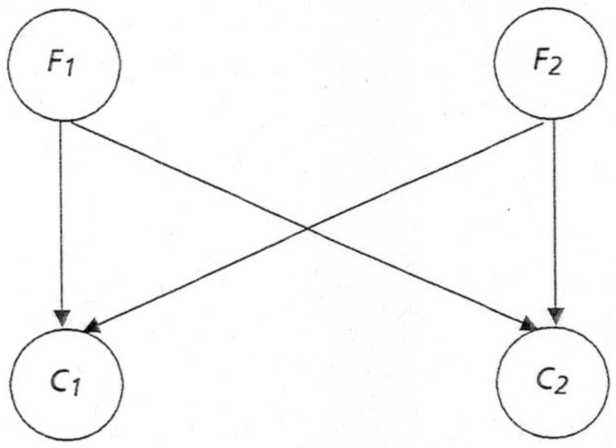


Figure 4.3.3

agent runs a transaction. These transactions are:

On fragment C_1: $[(T_{C_1}, w, c_{1,1})]$ (this customer wants to reserve a seat on flight 1)

On fragment C_2: $[(T_{C_2}, w, c_{2,2})]$ (this customer wants to reserve a seat on flight 2)

On fragment F_1: $[(T_{F_1}, r, c_{1,1}), (T_{F_1}, w, f_{1,1}), (T_{F_1}, r, c_{2,1}), (T_{F_1}, w, f_{2,1})]$

On fragment F_2: $[(T_{F_2}, r, c_{1,2}), (T_{F_2}, w, f_{1,2}), (T_{F_2}, r, c_{2,2}), (T_{F_2}, w, f_{2,2})]$

Here is the schedule: $(T_{F_2}, r, c_{1,2})$

$(T_{F_2}, w, f_{1,2})$

$(T_{C_1}, w, c_{1,1})$

$(T_{F_1}, r, c_{1,1})$

$(T_{F_1}, w, f_{1,1})$

$(T_{F_1}, r, c_{2,1})$

$(T_{F_1}, w, f_{2,1})$

$(T_{C_2}, w, c_{2,2})$

$(T_{F_2}, r, c_{2,2})$

$(T_{F_2}, w, f_{2,2})$

The precedence order between actions originating at different nodes is determined by the time when updates are installed at remote copies.* For instance, the fact that action $(T_{C_1}, w, c_{1,1})$ precedes action $(T_{F_1}, r, c_{1,1})$ in the schedule means that the update to $c_{1,1}$ by transaction T_{C_1} was installed in the copy at the home node of $A(F_1)$ before transaction T_{F_1} read this data object.

It is not difficult to see that the above schedule is not serializable in the usual sense, but it is fragmentwise serializable. In a conventional system this schedule would be prevented. For instance, $(T_{C_2}, w, c_{2,2})$ might be delayed till T_{F_2} was completed, reducing availability. In that case, the net difference between two executions would be that the serializable one would result in not reserving seats for the second customer on flight 2 (this would be corrected eventually since agents $A(F_1)$ and $A(F_2)$ run their transactions periodically). Thus, in this example replacing global serializability by fragmentwise serializability did not result in any serious anomalies while allowing for more flexibility in scheduling.

In summary, fragmentwise serializability is very simple but, we believe, powerful concept. Each fragment is treated as an independent database. Data read from other fragments may reflect non-serializable anomalies. However, the transactions operating on this fragment can cope with these anomalies and guarantee fragment consistency. The situation is analogous to dealing with user inputs in conventional databases. (For instance, banks never assume that users and data outside the system are handled in a serializable fashion.) Fragmentwise serializability lies somewhere in the center of our intuitive correctness - availability spectrum.

----

*See [11] for an in-depth discussion of the notion of transaction schedules in distributed systems.

4.4. Moving agents.

Allowing agents to move from node to node can be desirable for a number of reasons. For example, a bank customer would certainly like to be able to use the automatic teller machines at more than just one location. As another example, consider an airline database where there is a special fragment for seat assignments on a flight. Suppose there is a computer node at every airport and consider a flight which has stop-overs. (Passengers can be discharged and taken at each stop.) It would be desirable, for maximum availability, to make the computer at the airport where the flight is making a stop the current agent for the seat assignment fragment (initially the agent is the airport where the flight originates). Note that in this example the plane can be viewed as a token for the seat assignment fragment. Another reason for moving agents is node failure. When an agent's home node goes down, the agent may wish to re-attach to some other node in the network rather than wait until its home node comes back up.

Allowing agents to move, however, may endanger not only fragmentwise serializability but also, in some cases, mutual consistency of replicas. Let A be an agent that decides to move from node X to node Y (Figure 4.4.1). Let T_1 be the last update transaction initiated by A at X, T_2 the first update

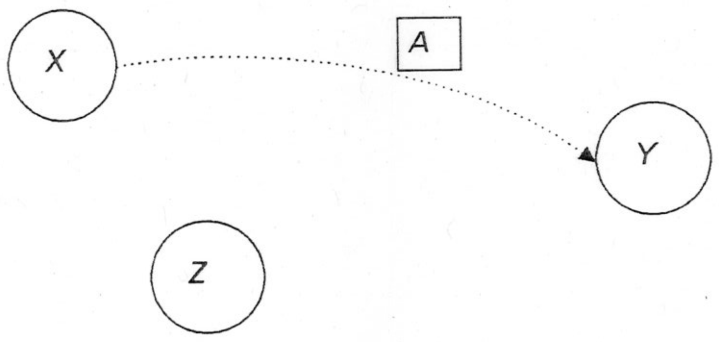


Figure 4.4.1.

transaction initiated by A at Y after the move has completed. In the absence of any special provisions, it is possible for T_2 to be initiated before T_1 has a chance to reach Y (if, for example there was a break in communications between X and Y). It is also possible for T_2 to be received at some other node, say Z, before T_1 is received there. It should be clear that such events may lead to violations of fragmentwise serializability and even mutual consistency. In both cases, transaction T_1 was delivered late (out of order). Therefore we refer to this as the problem of *missing transactions*. It is impossible, generally speaking, to circumvent this problem without paying a price in availability.

There are a number of ways to cope with missed transactions. They seem to fall into three categories. In the first, certain actions are undertaken by the system on a permanent basis, allowing agents to complete their moves "smoothly." The second category provides for some special actions by the system only at the time of a move. Finally, in the third category, agents are allowed to move without any preparatory actions; however, some actions might be taken after the move to rectify possible inconsistencies. There is a very large number of actual protocols for moving agents. We do not attempt here to be complete, nor do we strive for presenting the most efficient protocols. What follows is just a sample of possibilities, which hopefully gives a flavor of the issues involved.

4.4.1. Permanent preparatory actions.

The following method is suitable for those cases when an agent has to leave its home node, for whatever reason, and it is not particularly important exactly which node it is going to move to. This method uses a majority commit protocol. Before a transaction can commit at the agent's home node, the corresponding quasi-transaction is sent out to the rest of the nodes, and acknowledgments are requested. The transaction commits only after acknowledgments have been received from a majority of the nodes. Then a command is broadcast to commit the quasi-transaction at remote nodes. When the agent needs to move, a new home node is selected. (The selected node is the one with the token. Note that if the token was lost because of a failure, it can be reconstituted through an election.) The agent must then contact a majority of nodes and request an identifier for all previously executed quasi-transactions on the fragment. If the new home node had missed any of these, it requests them from the nodes that have them and runs them. This procedure ensures that the home node has seen all transactions previously executed on the fragment. (Each old transaction was seen by a majority of nodes. This guarantees that at least one node in the current majority has seen it and gives it to the new home node.) Now the agent is ready to execute new update transactions. The first of these receives the sequence number that follows the last transaction, and so on, so that there is a single, uninterrupted sequence of transactions. (As usual, other nodes execute the corresponding quasi-transactions in this same order.)

It is not difficult to see that fragmentwise serializability (or global serializability if there are additional restrictions on read accesses) is preserved with this method. However, the communication overhead can be significant, and of course, availability is reduced. Specifically, update transactions can only be processed with the cooperation of a majority group of nodes.

4.4.2. Actions at the time of the move.

A. Moving with data.

Let A be the agent that moves from node X to node Y. Let T_1 be the last update transaction initiated by A at X, T_2 the first update transaction at Y. We require that A transport (by any means available) a copy of the fragment stored at X to store it in place of the copy of the fragment at site Y before resuming processing. In addition, all other sites are requested not to install updates from transaction T_2 until those from T_1 have been installed. This method guarantees that not only mutual consistency but also fragmentwise serializability are preserved.

The requirement concerning transportation of data by the agent is not as difficult as it may sound at first. Consider the following example. A military command post is to be evacuated to avoid the threat of an enemy attack. The data from the fragment controlled by it can be dumped on tape and transported to a new location. This guarantees that the copy of the fragment at the new command post is now up-to-date. As another example, identification cards with magnetic strips are now becoming common. The strips in many cases can carry, not just an identifying code, but also real data (readable *and* writable). For instance, subway cards record the amount of money a user has available; copier cards store the number of copies a person is authorized to make. These cards fit our model exactly. As the agent moves, it carries with it a copy of the fragment it controls.

If the transactions that update a fragment must also read it, it is important that the agent has access to the most up-to-date version of the fragment. However, in those cases when all transactions are write-only, there is no need to transport data. Moreover, when these transactions are commutative (such as incrementing or decrementing some values, or creating new data items), copies of the fragment at different nodes will be mutually consistent regardless of the order in which they receive these updates. Hence, fragmentwise serializability is preserved. This was the situation in the banking example of Section 2. There, deposits and withdrawals simply entered records into the ACTIVITY fragment. Hence, customers (ACTIVITY agents) could move around without compromising correctness. (This assumes that withdrawals read the BALANCE fragment to make a decision, and not the ACTIVITY fragment itself.)

B. Moving with the sequence number.

With this approach, only the sequence number of the last transaction to run at the old home node is given to the new home. Let A, X, Y, T_1, and T_2 be as above. Before A can execute T_2 at Y, it must receive the sequence number of T_1 from X. This number can either be explicitly requested after the move or can be carried by the agent (*e.g.*, in the magnetic strip of a card). Before A executes T_2, it must wait until all previous quasi-transactions are received and run at Y. New transactions are given sequence numbers that follow that of T_1.

This method, just as moving with data, preserves fragmentwise serializability. It may result, however, in decreased availability due to the waiting for old transactions. However, it may be easier to implement.

4.4.3. Omitting preparatory actions.

In those cases where the previous approaches are infeasible, the only option left is to do the best we can after the move in order to minimize the effects of missed transactions. As before agent A moves from node X to node Y. Let $T_1, \ldots, T_r$ be the transactions initiated by A at X, and $\mathbf{T}_1, \ldots, \mathbf{T}_s$, the transactions initiated by A at Y, after the move. We assume that it is imperative that A start processing new transactions as soon as it arrives at Y. In such a situation, fragmentwise serializability can be compromised. However, at least we can guarantee that mutual data consistency is preserved, by enacting the following protocol.

Let $T_i \, (i \leq r)$ be the last transaction from X installed at Y before A initiates $\mathbf{T}_1$. Let Z denote any node in the system different from X and Y. Further, let us assume, that transactions and data items are timestamped.

A. *<u>At node Y</u>:*

(1) Before broadcasting $\mathbf{T}_1$: Broadcast a special message $\mathbf{M}_0 = (T_1, \ldots, T_i)$ (containing all transactions from X installed at Y thus far).

(2) Upon receipt of quasi-transaction $T_l \, (i < l \leq r)$ after $\mathbf{M}_0$ was broadcast (T_l is a missing transaction that has just been found): Remove from T_l those updates to items that have already been overwritten by more recent transactions. Package the remaining updates into a new transaction $\mathbf{T}_\mathbf{k}$, where $\mathbf{k}$ is the next sequence number. Install the updates of $\mathbf{T}_\mathbf{k}$ locally and send $\mathbf{T}_\mathbf{k}$ out as a regular quasi-transaction. If this missing transaction causes an anomaly that can be detected, then issue the necessary corrective actions. (For example, if after $\mathbf{T}_\mathbf{k}$ runs, a flight is overbooked, then cancel one or more reservations.)

B. At node Z:

(1) Upon receipt of $\mathbf{M_0}$: (Let T_j $(j \leq r)$ be the last transaction from X installed at Z.) If $j < i$, install transactions $T_{j+1}, \ldots, T_i$.

(2) Upon receipt of quasi-transaction T_q $(max(i, j) < q \leq r)$ *after* $\mathbf{M_0}$ *was received* (T_q *is a missing transaction*): Do not process T_q. Instead, forward it to Y, so it can take corrective actions.

(3) Upon receipt of quasi-transaction $\mathbf{T_q}$ $(q \geq 1)$: Process it as usual, after installing $\mathbf{T_{q-1}}$.

It is not hard to verify that all copies of data will eventually converge. This method for moving agents is somewhat similar to the "free-for-all" systems, in that it gives high availability (the agent can start processing transactions as soon as it arrives at the new home node), and the correctness criterion is weak. The advantage of this method over "free-for-all" systems is that all decisions concerning corrective actions for a fragment are centralized.

6. Conclusions.

We have presented a family of strategies for obtaining high data availability in a distributed system. Within the same framework of fragments and agents we can obtain, as discussed, global serializability, fragmentwise serializability, or simple mutual consistency.

The basic idea of our approach is quite simple: give control of the data to the users or nodes who actually need it. This avoids many synchronization conflicts, while at the same time making data available to those who need it.

Due to space limitations, there are a number of interesting issues that are not covered here. As we have stated, it is essential to design the database correctly, if one is to fully take advantage of our approach. In [7], we have outlined some guidelines for good design. These include ways for properly partitioning the database into fragments, strategies for avoiding multi-fragment transactions, and techniques for minimizing the number inter-fragment consistency constraints.

Our approach can be generalized for dealing with transactions that update multiple fragments and with databases that are not fully replicated. Finally, it is also possible to combine several of our strategies in a single system. Since all of our strategies are based on the same framework, this combination is not difficult. Hence it is possible to guarantee mutual consistency for some fragments (with the mechanism of Section 4.4.3, say), fragmentwise serializability for a set of other fragments (with any of several techniques), and conventional serializability within another group (by having read-access restrictions, say). This gives us even greater flexibility in tailoring a system to the correctness and availability requirements of the users.

7. Bibliography.

[1] Blaustein, B.T., H. Garcia-Molina, D.R. Ries, R.M. Chilenskas, and C.W. Kaufman, "Maintaining Replicated Databases Even in the Presence of Partitions," *Proc. IEEE EASCON Conference*, 1983.

[2] Blaustein, B.T., C.W. Kaufman, "Updating Replicated Data During Communications Failures," *Proc. 11th VLDB*, 1985.

[3] Bernstein, P.A., N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981.

[4] Davidson, S.B., "Optimism and consistency in partitioned database systems," *ACM Trans. Database Syst.*, Vol. 9, Num. 3, September 1984.

[5] 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.

[6] Garcia-Molina, H., T. Allen, B. Blaustein, R.M. Chilenskas, and D.R. Ries, "Data-Patch: Integrating Inconsistent Copies of a Database after a Partition," *Proc. 3rd IEEE Symp. on Reliability of Distributed Software and Database Systems*, October 1983, IEEE, New York.

[7] Garcia-Molina, H., and B. Kogan, "Achieving High Availability in Distributed Databases (Full Version)," Technical Report (in preparation), Department of Computer Science, Princeton University.

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

[9] Minoura, T., and G. Wiederhold, "Resilient Extended True-Copy Token Scheme for a Distributed Database System," *IEEE Transactions on Software Engineering SE-8*, 3, May 1982, 173-189.

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

[11] Traiger, I.L., J. Gray, C.A. Galtieri, and B.G. Lindsay, "Transactions and Consistency in Distributed Database Systems," *ACM Trans. Database Syst.*, Vol. 7, Num. 3, September 1982, 323-342.

8. Appendix.

(This appendix contains a proof of the theorem of Section 4.2. If this paper is accepted for publication, we plan to delete this appendix and refer the reader to our extended report [7]. In other words, we include this proof here just in case the referees wish to see it. However, if the referees want it in the final version and there is space for it, we will be glad to leave in.)

Definition 8.1. A transaction T_j is of type F_i ($tp(T_j)=F_i$) if T_j is initiated by $A(F_i)$.

Definition 8.2. The *global serialization graph (g.s.g.)* is a directed graph whose vertex set is the entire set of transactions executed by the system and whose set of edges is computed according to the following rules:

(i) For two transactions T_i and T_j such that $tp(T_i)=tp(T_j)$, it is determined whether there is a directed edge between them according to the standard dependency rules for centralized databases.

(ii) Let $tp(T_i)=F_q$, $tp(T_j)=F_r$ $(q \neq r)$, and let (F_q, F_r) be an edge in the read-access graph. Then if there is a data item $d \in F_r$ that is read by T_i and updated by T_j, and the update to d produced by T_j is installed in the copy of the database at the home node of $A(F_q)$ before T_i reads d, put in edge (T_j, T_i); if the update is installed after T_i reads d, put in edge (T_i, T_j).

Definition 8.3. The *local serialization graph (l.s.g.)* for fragment F_i is a directed graph whose vertex set contains all transactions T_j such that $tp(T_j)=F_i$ or $tp(T_j)=F_s$ and (F_i, F_s) is an edge in the read-access graph. Its edges are computed according to the following rules:

(i) For two transactions T_l and T_j, such that $tp(T_l)=tp(T_j)=F_i$, it is determined whether there is a directed edge between them according to the standard dependency rules for centralized databases.

(ii) Edges between a local transaction (of type F_i) and a non-local one are computed as in (ii) of Definition 8.2.

(iii) For a pair of non-local transactions of the same type, T_l and T_j, put in edge (T_l, T_j) if T_l is installed at the home node of $A(F_i)$ before T_j, edge (T_j, T_l) otherwise.

(iv) If T_l and T_j are non-local transactions of different types, then there is no edge between them.

Theorem. The global serialization graph is acyclic if all the local serialization graphs are acyclic and the read-access graph is elementarily acyclic.*

Proof.

We assume that local transactions executed at the same node are broadcast in the order imposed by the relevant l.s.g. The broadcast mechanism, then, ensures that they are received at remote sites in the same order.

To simplify the proof we assume that the number of fragments equals the number of nodes in the system (n), every agent controls exactly one fragment, and, finally, there is exactly one agent at every node. Later on we shall show how to lift these restriction preserving the result of the theorem.

The proof uses induction on n. For $n=1$ there is only one l.s.g. and it is equivalent to the g.s.g., thus the theorem trivially holds. Suppose it also holds for some n-1 and consider a database consisting of n fragments, with the read-access graph (call it R^n) satisfying the condition of the theorem. Since R^n is elementarily acyclic, there must be a vertex in it, say F_q, which is a head or a tail of only one edge. Consider the following two cases.

Case 1. F_q is the tail of a single edge (and there are no edges incident upon F_q). Let F_r be the head of this edge (see Figure 8.1). Let R^{n-1} denote the graph that results from R^n after deleting F_q and the edge that emanates from it. R^{n-1} is the read-access graph for a database consisting of n-1 fragments. Let G^n denote the g.s.g. of the original database. Similarly, let G^{n-1} denote the g.s.g. of the database with fragment F_q removed, that is, with all transactions of type F_q deleted. Finally, let $L^n(F_i)$ be the l.s.g. for fragment F_i, $i=1, ..., n$, in the original database, and $L^{n-1}(F_i)$, the l.s.g. for fragment F_i, $i=1, ..., q-1, q+1, ..., n$, in the reduced database. Notice that $L^n(F_i)=L^{n-1}(F_i)$, for $i=1, ..., q-1, q+1, ..., n$. This is true because there are no read-access edges incident *upon* the removed fragment F_q. Hence, $L^n(F_i)$ ($i \neq q$) cannot have any vertices representing transactions originating from $A(F_q)$ (see Definition 8.3).

----

*Note that this theorem was formulated in slightly different terms in Section 4.2.
It is intuitively clear, and can be proven rigorously, that acyclicity of the g.s.g. is equivalent to serializability of the global transaction schedule in a distributed database system. Also,local concurrency control mechanisms will guarantee that all the l.s.g.'s are acyclic.

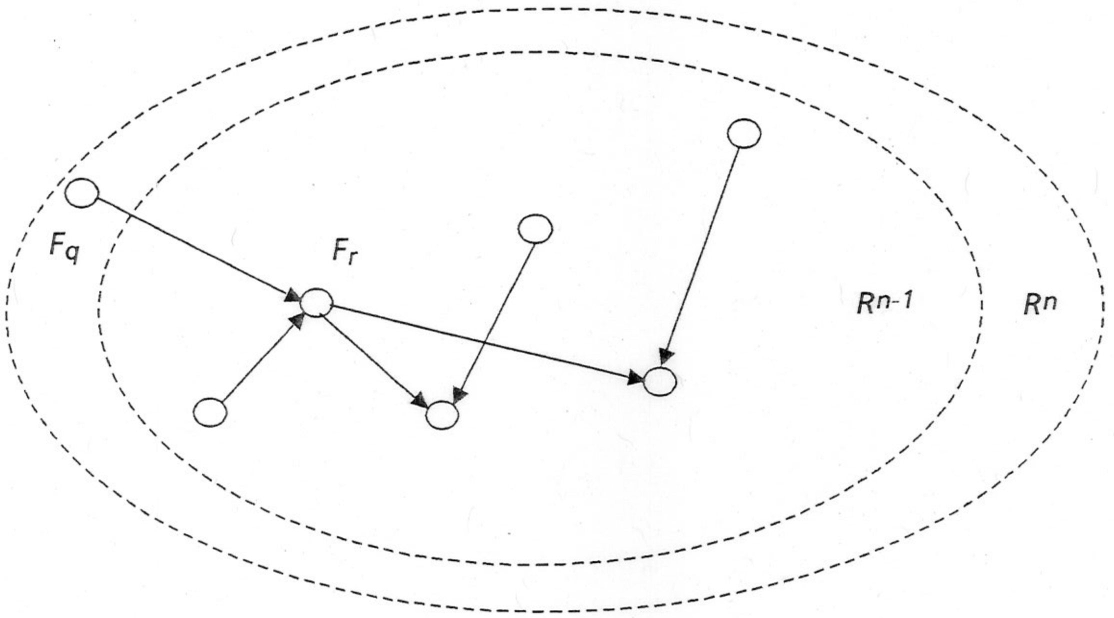


Figure 8.1.

If G^n had a cycle containing no transactions of type F_q, G^n would also have a cycle. However, our induction hypothesis tells us that if R^{n-1} is elementarily acyclic and all $L^{n-1}(F_i)$ $(i \neq q)$ are acyclic, then G^{n-1} is acyclic. Since both conditions are true ($L^n(F_i) = L^{n-1}(F_i)$ $(i \neq q)$, which is acyclic), then we have a contradiction.

So we can assume that any cycle in G^n contains at least one transaction of type F_q. Let $C = (T_1, ..., T_k, T_1)$ be one such cycle (see Figure 8.2). Let u be the number of paths on this cycle consisting exclusively of transactions of type F_q (in Figure 8.2, $u=3$). For $j=1, ..., u$, let T_{l_j} be the vertex on C that immediately precedes the j-th path, and T_{m_j}, the vertex that immediately follows it. Since F_r is the only vertex in R^n that shares an edge with F_q, we can conclude that $tp(T_{l_j}) = tp(T_{m_j}) = F_r$, for all $j = 1, ..., u$. Finally, let P denote the collection of (disjoint) paths on C consisting of transactions that are not of type F_q.

We will prove, by induction on u, that P cannot be totally contained in $L^n(F_r)$. First, let $u=1$. Then G^n contains a path from a path $X = (T_{l_1}, T_{h_1}, ..., T_{h_g}, T_{m_1})$, where $tp(T_{h_1}) = \ ... \ = tp(T_{h_g}) = F_q$ and $tp(T_{l_1}) = tp(T_{m_1}) = F_r$. In this case, P consists of just one path, a path from T_{m_1} to T_{l_1}. Now, let us look at $L^n(F_r)$ and assume that P is contained in it. Recall that $L^n(F_r)$ represents the dependencies at the home node of $A(F_r)$. Thus, the existence of P in $L^n(F_r)$ implies that T_{m_1} must have been executed by

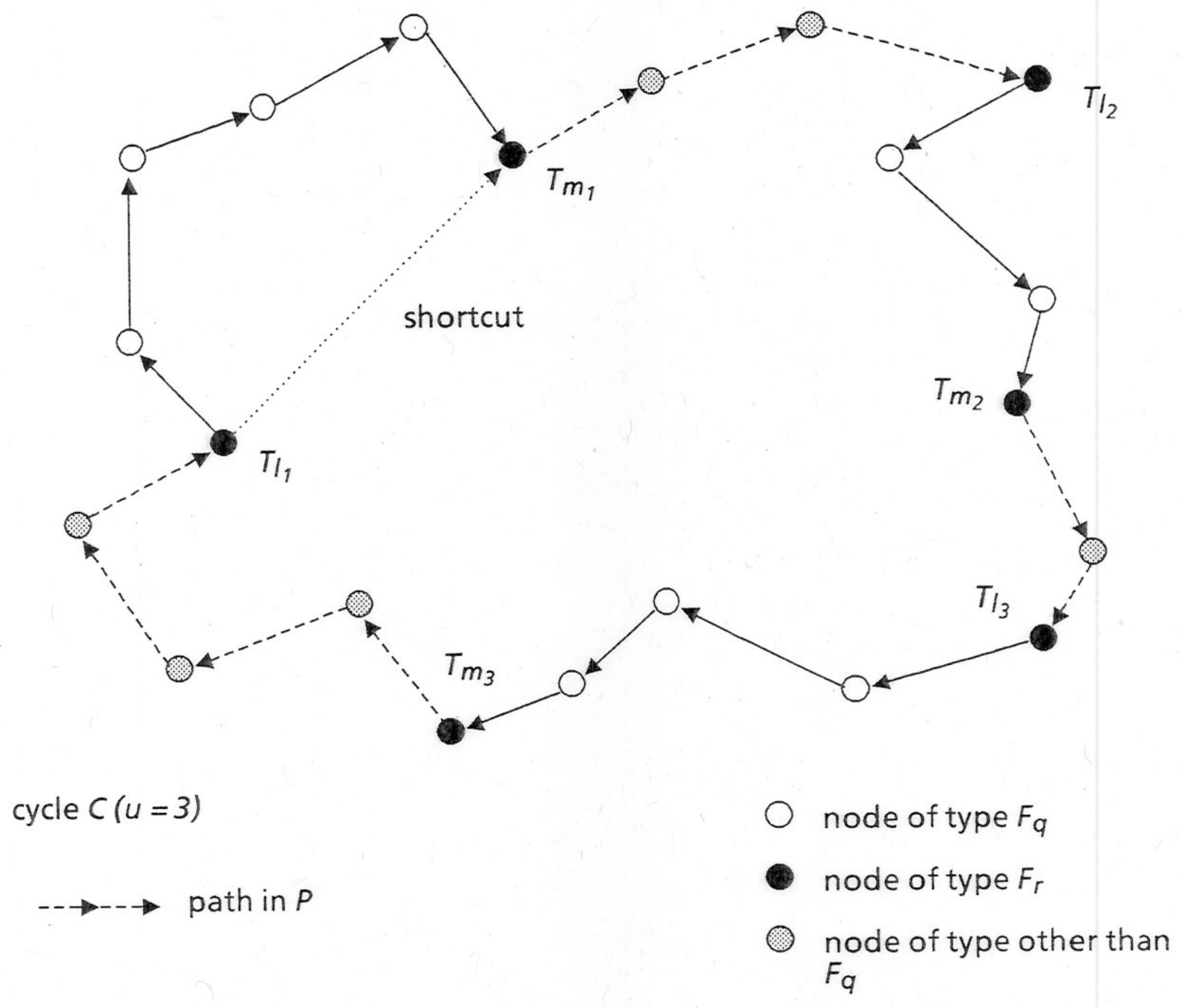


Figure 8.2.

$A(F_r)$ before T_{l_1}. This, in turn, implies that their updates at the home node of $A(F_q)$ were installed in this same order. Therefore, there is an edge (T_{m_1}, T_{l_1}) in $L^n(F_q)$. Furthermore, X is totally contained in $L^n(F_q)$, so $L^n(F_q)$ has a cycle, a contradiction. Thus, for $u=1$, P cannot be totally contained in $L^n(F_r)$.

Proceeding by induction, suppose that for some u-1, P cannot be totally contained in $L(F_r)$. Assume C has u paths on it consisting exclusively of transactions of type F_q. As before, let X be one of these paths. We will use cycle C to construct a shorter cycle that cannot exist. This will prove that C itself does not exist. Let us introduce a new data item into fragment F_r, *i.e.*, let us replace F_r by $F_r^* = F_r \cup \{a\}$, where $a \notin F_r$. Also, let us introduce in the transaction schedule at the home node of $A(F_r)$ two new actions, one belonging to T_{l_1} and the other, to T_{m_1}, in the order given below:

$$(T_{l_1}, r, a)$$

$$(T_{m_1}, w, a)$$

This will create an edge (T_{l_1}, T_{m_1}) in G^n. The same edge will also appear in $L^n(F_r)$ (if it was not there before) but it will not create a cycle in it. (If it did, then there would be a cycle in G^n containing only one path consisting exclusively of transactions of type F_q $(u=1)$, which was shown not to be possible.) Let us call the trick of creating a new edge between two transactions by means of adding a new data item *shortcutting*. In this case, when we shortcut from T_{l_1} to T_{m_1}, we end up with a cycle in G that contains u-1 paths of the above mentioned kind, which contradicts the induction hypothesis. Thus, we conclude that C cannot exist.

So we are forced to assume that, if a cycle C exists in G^n, P is not totally contained in $L^n(F_r)$. Let us consider, once again, the reduced database, without fragment F_q. C loses all transactions of type F_q, and each gap in C is bounded by nodes of type F_r. Therefore, we can shortcut all such gaps, which produces a cycle in G^{n-1}. At the same time, we know that $L^{n-1}(F_i)$, $i \neq q, r$, are still acyclic. Furthermore, $L^{n-1}(F_r)$ is also acyclic, in spite of the shortcuts we have introduced. (In particular, since P is not contained in $L^{n-1}(F_r)$, the new cycle will not be in $L^{n-1}(F_r)$ either.) In addition, R^{n-1} is elementarily acyclic. Thus, our inductive hypothesis implies that G^{n-1} is acyclic. This contradiction means that G^n is acyclic. This completes the inductive proof.

Case 2. Assume that F_q is the head of the only edge that touches it (G^n looks just like in Figure 8.1, except edge (F_q, F_r) is replaced by edge (F_r, F_q)). Let C be a cycle in G^n, and let P be as before. Since in this case $L^n(F_r)$ contains all transactions of type F_q, P cannot be totally contained in $L^n(F_r)$ (otherwise P plus transactions of type F_q would complete cycle C within $L^n(F_r)$). Thus, if we leave out fragment F_q and shortcut the gaps in C, we will have an acyclic $L^{n-1}(F_r)$, whereas G^{n-1} will have a cycle, which contradicts the induction hypothesis.

In the beginning of the proof, we made an assumption that there is one fragment per agent per node. If an agent controls more than one fragment, in reality, the agent can be split conceptually into several agents controlling just one fragment each. Similarly, when more than one agent share the same home node, this node can be viewed, for the purposes of the proof, as several nodes housing just one agent each. This completes the proof of the theorem.