DATA-VALUE PARTITIONING AND VIRTUAL MESSAGES*

Nandit Soparkar and Abraham Silberschatz

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-19 July 1989

ABSTRACT

Network Partition failures in traditional Distributed Databases cause severe problems for transaction processing. In this paper, we observe that the only way to overcome the problems of 'blocking' behavior for transaction processing in the event of such failures is, effectively, to execute them at single sites. A new approach to data representation and distribution is proposed and it is shown to be suitable for failure-prone environments. We propose techniques for transaction processing, concurrency control and recovery for the new representation. Several properties that arise as a result of these methods, such as non-blocking behavior, independent recovery and high availability, suggest that they could be profitably implemented in a distributed environment.

Keywords: Distributed Databases, Transaction Processing, Fault Tolerance, Concurrency Control.

*This material is based in part upon work supported by the Texas Advance Research Program under Grant No. 4355 and by the NSF under grant IRI-8805215.

1 Introduction

Traditional distributed database systems which have concurrently executing transactions require specialized protocols to ensure the consistency of the data stored [1]. The design of such protocols in the case where either the transactions running at the sites fails, or the communication channels fail, is quite complicated. Among the most severe kinds of failures that are encountered in practical situations is the *network partition* failure [3]. Handling such failures, given the traditional manner of representing data is, in a sense, an intractable problem [8].

Network partitions result when a system of sites connected by communication links gets separated into more than one group such that different groups are unable to communicate. This is caused by the failure of communication links, sites or both. The difficulties encountered in such situations is two-fold. Transactions in progress when the failure occurs need to be held-up as they are not able to commit or abort safely [3, 8]. This results in the non-availability of data locked by these transactions. The second major difficulty arises in connection with transaction processing after the network has partitioned. If there is replicated data, as is often the case, it is unsafe to allow changes to be made to locally available copies, as the changes are not available at all the sites. Even if data is not replicated, some transactions which could have been safely run, may not be allowed to do so since the copies of the data items in question are not available.

This scenario becomes more complicated in the realistic situation of several partitions occurring at the same time, the network getting restored, and recovery processes executing in a dynamic environment. In fact, when we do not use fail-stop processes (even without allowing 'malicious' behavior), the concept of a network partition itself becomes difficult to define. The meaning of a partition that is not 'clean' is unclear [3]. In such a situation, the meaning of detection of failures is also imprecise.

In this paper we propose a new approach to distributed transaction management to alleviate these problems. The key idea is to view the data in a different manner as compared to traditional systems. It will become apparent that handling such data will also entail non-traditional approaches to data manipulation. We feel that the ideas presented here are sufficiently novel to warrant the coining of the terms *Data-value Partitioning* and *Virtual Messages* (henceforth referred to by DvP and Vm, respectively).

The rest of the paper is organized as follows. In Section 2 we discuss the problems encountered due to network partitions and conclude that the only effective solution is to run each transaction at one single site with the possibility of some cooperation from other available sites. The ideas of data-value partitioning and virtual messages are introduced in Section 3. The subsequent three sections deal with the related issues of transaction processing, concurrency control and recovery. The penultimate section deals with the more apparent merits and demerits of the approach suggested in this paper.

2 Problems Associated with Network Partitions

In this section we examine the two problems outlined in the introduction more closely. We note that the problems we discuss here pertain to traditional distributed databases. We present some of the pertinent results from our work in [9] where the reader can find the formal model and the proofs for these results.

We assume that there are n sites which are linked by communication links in some interconnection topology. We are interested in a *commit* protocol where each site is considered to execute a process which can be modeled as an automaton. Each site, upon activation of the commit protocol, executes a process corresponding to its automaton. A commit protocol is said to be *non-blocking* with respect to some set of possible failures in the system if every participating site reaches a decision (to commit or to abort) in a bounded number of steps (as measured locally), regardless of the type of occurring failures [8, 4, 9]. A commit protocol is said to be *blocking* if it is not non-blocking. A protocol, or a transaction making use of a protocol, is said to *block* if, for some reason, it is not able to complete in the bounded number of steps that are assigned to it. In such situations it is typically the case that the site executing the blocked protocol or transaction is awaiting the occurrence of some event such as the reception of a message from another site. The result is the wasteful consumption of resources such as computation time and data accessibility, and results in holding-up the processing of further transactions.

Non-blocking protocols are of interest for many realistic applications since a bound can be placed on how long they will take to complete. Clearly, this is important for database applications where resources that are being used by some transaction should be made available for other transactions as soon as possible. Furthermore, it is often desirable to get quick responses to queries and this could be enforced by using the bounds available on the completion times. In this paper, we restrict attention to non-blocking protocols.

2.1 Non-Existence of a Commit Protocol

From the results in [8, 9], the non-existence of a commit protocol in several important situations is well established. It is especially significant to consider that there are no commit protocols in the case where the system can suffer from network partition failures. Relevant details may be obtained from [9].

The implications of this result are very important. If there is a transaction executing at more than one site in a traditional database, and the transaction has to update data at several locations, then there is always the possibility of a partition failure that will block the transaction. We need to state that there be several sites at which updates need to be made, since at sites where updates are not necessary (i.e., sites that have read-only data), it may be possible for the concurrency control scheme to release the read locks on the data by some time-out mechanism. However, as far as recording updates is concerned, for obvious reasons, *all* or *none* should be permitted. There are other ways which do not require that all the writes occur, but they require something similar to having a majority of the sites implementing the write (if replication of the data is assumed) which leads to the same problems.

The blocking behavior follows due to the requirement that a transaction either commits at all locations, or aborts at all locations [2]. Hence, *the only way that transaction processing could be guaranteed to exhibit non-blocking behavior is if a transaction writes data at only one site.* It is not difficult to see that there *are* ways to exhibit such behavior if transactions are so restricted — specifically, some techniques used for databases that are not distributed are examples. However, the concurrency control aspects and how data can be obtained from other sites are questions whose solutions are not altogether obvious. We shall examine some methods to achieve these ends in this paper.

2.2 Partitioned Operation in Traditional Systems

Suppose that by some stroke of good fortune the network cleanly partitions while no transactions are in progress. Even in this case, the problem of how to continue processing new transactions has no simple solution. Several techniques, both 'syntactic' and 'semantic', have been proposed [3]. Syntactic approaches use serializability as the criterion for maintaining correctness of the stored data. In general, these methods must be used if 'irreversible' transactions (i.e., transactions on which no reasonable corrective action can be effected) could occur. An example of this is a transaction that disburses money to a customer in a banking application. Semantic methods make use of the contents of the database. That is, by making use of specific properties of the data, serializability is sacrificed and the criterion considered for correctness is some other aspect (such as consistency of data between the partitioned groups after the failure is rectified). In many of the semantic approaches, the data stored in different partitions are allowed to become mutually inconsistent. Later, after the partition has been rectified, an attempt is made to correct the inconsistencies. It should be noted that such approaches cannot be applied in situations such as banking applications where data consistency is critical.

After a partitioning has occurred, several restrictions have to be imposed to ensure the correct execution of the system. If a data item is replicated, and copies exist in more than one group of the partition, then either a single partition is allowed to access the item to read and write (implemented by quorum, primary copy etc. [3]). If reading is permitted, then no site is allowed to update the item. In this context, in cases where a *read* is permitted for a data item in a partition, it should be known that a *write* on the same item is not permitted in some other partition [9]. Furthermore, it is not always possible to ensure that a single group accesses the item (e.g., a quorum is not reached, or a primary copy site fails). Even in the case that data items are not replicated, it is possible that some 'safe' transactions are not allowed. For example, if an individual's account balance in a banking environment is inaccessible due to a network partition failure, then if the person wants to deposit some money (without caring about the net balance) this is not possible as the balance is not accessible. Several methods have been suggested to alleviate this problem [3] but their usefulness is limited by the fact that many irreversible transactions may not be permissible.

A related consideration that compounds the problems is that it is, in general, not possible to determine partition failure characteristics [9]. Consider, as an example, two sites communicating via a link. Assume that both sites and links are failure-prone and have otherwise 'clean' behavior and favorable synchronism characteristics. If the communication link fails at some time and no undeliverable messages are returned, then either site is unable to determine whether the remote site failed, the link has failed, or both did. Note that in such situations, the only way a site determines that something is amiss is by a time-out mechanism when an expected message fails to arrive. No partition detection algorithm can be expected to handle such general situations and, as such, transaction processing techniques which rely on the detection of partitions are not robust. Methods that have been suggested for techniques to handle partition failures are very often quite complicated, computationally expensive, or do not handle the more general cases of partitions.

3 The New Approach

We have advocated in the conclusion of Section 2 that a new approach to distributed transaction management must be developed if non-blocking protocols are to be obtained. As a step in this direction, we examine a novel approach in this paper. Our methods apply essentially to data that can be partitioned into smaller pieces such that the pieces themselves can be regarded as instances of the original data items. We clarify this with a simple representative example. Consider a data item denoting some amount of money. We can split the amount into several smaller portions. Each of these portions are themselves some amounts of money. The same operators that applied to the original data item also apply to the separate portions (e.g., increment, decrement, set to zero, etc.). Also, adding two portions together yields another portion which is again an instance of an amount of money. Notice that we regard the data in a manner similar to the way in which resources are regarded in an operating system.

Our approach is based on two key ideas. The first idea is to split-up the values of the data items that are stored in the database. For example, the number of seats in airline reservation systems, the number of units of an item in an inventory control system, or the amount of money in the bank balance of an individual for banking applications. These correspond to the pieces of data described above. Each of the constituent values is stored at different sites. Every transaction that is to be run in the system, is executed at one single site by using data values available *at the same site*. Only in the event that the locally available data is inadequate for the transaction to execute, the site requests other sites for the values stored in them. Hence, the model of a system that we propose has transactions executing at single sites with only locally stored data and making infrequent requests to other sites in the case of being unable to proceed with what is available. If the responses from the remote sites fail to arrive for any reason within a reasonable amount of time, the transaction is aborted. Locks are required on the data values to be able to access them. The lock for a data value is obtained at the same site at which the data value is resident.

The second idea is to make use of a special method for communicating crucial data. By making use of stable logging facilities, it is ensured that this kind of crucial data is not lost. A more clear description of these methods is given ahead.

To illustrate these ideas, we present the following example. Consider a simple airline reservation system, where the system reserves seats for a flight A. Assume that seats can be reserved from four sites named W, X, Y and Z. These sites are expected to have sufficient computing and information storage facilities as needed. Let N represent the number of seats available at any given time on flight A. In our system, the sites W, X, Y and Z store values N_W, N_X, N_Y and N_Z, respectively, in connection with N. The relation between these is $N_W + N_X + N_Y + N_Z = N$. Initially, let $N = 100$ and $N_W = N_X = N_Y = N_Z = 25$. Note that N is *not* represented explicitly at any place.

Let the sites begin allocating seats to customers. If customers requesting 3, 4 and 5 seats arrive at site W and their requirements are met, the values that N_W takes on are 22, 18 and 13, respectively. These changes in the local value of N_W also affect the value of N since N is, in effect, *defined* by the values stored at the different sites. In case some customers cancel their seats, say at site W, then the number of seats that become available as a result of the cancellation are added to N_W. Note that in doing so, it could turn out that N_W exceeds the initial value of 25 that it had since customers reserving seats at some other site could

cancel their seat at site W. So far, we observe that sites need not have communicated with each other as far as the number of seats available, N, is concerned.

After some duration of time, the values stored at the different sites could be : $N_W = 2$, $N_X = 3$, $N_Y = 10$ and $N_Z = 15$. Thus, the total number of available seats on the flight A is given by $N = 2+3+10+15 = 30$. Now, suppose that a customer requiring 5 seats arrives at site X. Clearly, the request cannot be granted without increasing the value of N_X. The site X decides to request seats from some other sites. Depending on how the system is designed to function, a request for at least three seats is sent by site X to one or more sites among W, Y and Z. Once such a request has been initiated, a timeout mechanism at site X is triggered to await responses from the other sites. Of course, in the meanwhile, some other useful work can be done. The motivation for sending the requests is to redistribute the value of N among the sites so that the value of N_X is sufficient to handle the allocation of the required 5 seats.

At this point several different behaviors could be displayed by the system depending on several external factors, such as arrival of customers at different sites, or some internal factors such as faults occurring in the components of the system. Assume that site Z received the request sent by site X. Suppose that site Z decides to send 5 seats as a response. Site Z will decrement N_Z by 5 and send a message carrying a value of 5 to site X. In order to ensure that various system failures will not destroy the integrity of our scheme, additional mechanisms need to be introduced. In particular, the information concerning the new state of variable N_Z, and a copy of the message sent to site X must be recorded on a log residing in stable storage. Thus, in our example, the value of N_Z is reduced to 10 after a log record indicating this change *and* a message of value 5 to site X is recorded on stable storage. We assume that messages are numbered and that the failure of an acknowledgement by the intended recipient prompts a retransmission of the message. A more complete description of this scheme, which we refer to as Vm, is given in Section 4. By this mechanism, the data is sent by site Z to site X safely (that is, the data is not lost forever as a result of faults).

If all goes well and site X receives the message from site Z (and, perhaps, from other sites as well), the value of N_X is incremented by the amount of the data carried in the messages. For example, if site X received only the message from site Z, it increments N_X by 5. Clearly, the receipt and incrementing of the messages is done in a careful fashion involving logging of relevant data — the details are provided ahead in the portions dealing with the Vm. Once N_X is correctly updated, the request for 5 seats by the customer is handled in the normal manner.

Consider the situation at site X in the case that a timeout is signaled prior to the arrival of messages that make N_X sufficiently large to handle the required seat assignment request. In that case, the customer that needed 5 seats is not granted the request and the corresponding transaction is aborted. The reasons why no messages are received could be numerous : other sites may be malfunctioning, messages may be delayed, communication links may have failed, there may not be any more available seats, network partitioning may have occurred, and a host of other reasons. What the reasons may be is quite immaterial in our system. We do not require site X (or any other site) to attempt any detection or correction of such faults external to the site in the normal periods of processing.

Note that $N \geq N_W + N_X + N_Y + N_Z$ at all times. Also, if N_M represents the values in transmission, $N = N_W + N_X + N_Y + N_Z + N_M$. The need for a robust scheme of message transmission, such as the Vm, is not difficult to see. Also, if it is necessary to determine the value of N at a site, say site X, then it must be

ensured that $N_W = N_Y = N_Z = N_M = 0$. In that case, $N = N_X$ and the value may be determined easily. These aspects are discussed in detail in Section 5.

To be able to function usefully, the system makes use of transactions that can execute successfully using data at one site. In later sections it will become apparent that if data from several sites are needed, these are brought (by means of Vm) to the local site and processing is done at a single site alone. Also, operations that can be performed on the data during a partitioned state of the communication network must also have certain properties. We discuss these further ahead.

From the above descriptions, it is not difficult to see how the sites can handle reservations for several flights. There could be other flights B, C and D with corresponding values for the number of available seats P, Q and R. Transactions may involve more than one flight (e.g., a customer may want to change reservations from flight A to flight B). All such transactions are easily handled.

To summarize, let us consider the airline reservation system more abstractly. If N represents the number of seats available on a flight, we split-up the value of N and give a different quota to each site. We shall let site s_i be given N_i seats such that $\sum_i N_i = N$. Note that a traditional database without replicated data can be described trivially as a special case of this approach.

The system is expected to execute normal transactions such as reserving seats or canceling seats by making use of the local value of N. For example, reserving a seat at site s_i requires decrementing N_i. This can be done as long as N_i exceeds zero. In the case that the local value is insufficient to handle a transaction, requests are sent to other sites to obtain values with which the transaction successfully executes. In case the local site does not receive responses from other sites — which could be due to site failures, link failures or some other reason, the transaction is aborted. Essentially, the idea is that each site executes transactions using the values available to it locally as far as possible, and to request other sites for some values only when forced to do so.

It is clear that when failures do not occur, we can carry-out *read* and *write* operations by reading all the values and writing at least one of the values. In case of network partitions, each site is able to access at least its local quota and also any of the other site quotas that are accessible. This is clearly advantageous over the case that no processing can be done at all. Although this appears to be a rather simplistic notion, it will be shortly seen to have some nice properties. By making use of Vm, we shall be able to exhibit some non-blocking behavior.

Suppose that site s_i needs to send N_i to site s_j (for example, if site s_j needs to evaluate N). Site s_i, after locking N_i, writes one log record indicating that a message needs to be sent to site s_j with value N_i *and* that the N_i value at site s_i is now supposed to reflect the value 0. The idea is that the value of N_i is actually sent to site s_j (and removed from site s_i). After this, site s_i dispatches the requisite (real) message to site s_j and changes the local database to reflect the change in N_i. The lock on N_i may be removed now.

We note that in the above example, the allocation of a seat to a particular individual destroys the equivalence of the seat to every other seat since it is distinguished by the individual to whom it is allocated. Such seats, therefore, form separate data items which cannot be partitioned and have to be stored at one site (say the site where the allocation was made). If a cancellation were requested, a different kind of transaction would be needed, and that transaction may entail blocking behavior in the cases where some components of the system fail. However, in airline reservation systems, it is very desirable to have non-blocking allocation

schemes (to avoid losing a customer disgruntled by delays) even if the cancellation mechanisms exhibit blocking behavior. Here and below, it is to be understood that applications may require certain transactions that do exhibit blocking behavior although most types of transactions are non-blocking. We shall restrict our attention to partitionable operators and data (described below) that correspond to non-blocking transactions in our system.

4 System Structure

In this section, we examine the basic ideas of data-value partitioning and virtual messages. We also define the operators that can be used to manipulate the data in the system effectively. Once this is accomplished, it will become clear how the system is able to exhibit properties of non-blocking behavior and, at the same time, continue processing new transactions inspite of network partition failures.

4.1 Data-value Partitioning

Let d be a data item drawn from a domain Γ. Consider a non-empty multiset (i.e., a set with possible duplicates of elements) of values drawn from Γ denoted by b. The domain of the multisets is denoted by Γ^+. We say that a surjective mapping $\Pi : \Gamma^+ \rightarrow \Gamma$ denotes a *Data-value Partitioning* (DvP) of the elements of Γ. With some abuse of terminology, we shall let $\Pi^{-1}(d)$ denote the multiset b representing the partitioned values of d (where $d \in \Gamma$ and is stored in the system at that moment in the form of multiset $b \in \Gamma^+$).

To be useful, the DvP need to have some additional properties. The mapping Π must be easily computed. In addition, a *partitionable* property is required. Given a multiset $b \in \Gamma^+$ partitioned into multisets $b_1, b_2, \ldots, b_m$, where each $b_i \in \Gamma^+$, a multiset b' consisting of the elements $\Pi(b_1), \Pi(b_2), \ldots, \Pi(b_m)$ has the property that $\Pi(b') = \Pi(b)$.

We note that the airline seat allocation scheme of Section 3 meets these requirements. The domain Γ is given by the set of natural numbers, and the summation operator corresponds to the mapping Π. It should be clear that the DvP, Π, consists of the values N_i stored at sites s_i, and any Vm present in the system. The properties of summation provide us with the partitionable property for the seat allocation scheme.

Lastly, we need the concept of a 'partitionable' operator. We say that f is a *partitionable* operator for Γ and Π if its *effective* application to an element b_i of a multiset $b \in \Gamma^+$ results in a multiset b' such that $f(\Pi(b)) = \Pi(b')$. Ineffective applications result when, for reasons particular to the argument, the result is equivalent to a 'no-operation'. Examples of partitioned operators are 'increment the argument by m' and 'decrement the argument by m if the result does not fall below 0'. The latter operator exemplifies the need for 'effectiveness' in the definition above. We leave the exact nature of effective applicability to the reader as it should be intuitively clear what is being attempted. As we shall see, it is such partitionable operators that can be handled neatly in a non-blocking manner. Our concern is mainly with such operators.

As the system proceeds with its execution, it may happen that the multiset of values representing some data item changes while the value of the data item itself does not. Operators that cause such a change shall be called redistribution operators. That is, if $b \in \Gamma^+$, then h is a *redistribution* operator if $\Pi(b) = \Pi(h(b))$.

Let us provide the intuition behind the definitions. We require that the mapping Π be easily computed so that if the value of d, or the value represented by a portion of $\Pi^{-1}(d)$ were needed, then it would be found easily if the requisite multiset were available for the computation. In case of network partitions, the accessible values may be b_i, where b_i is one of the multisets into which $\Pi^{-1}(d)$ may be partitioned. It is then possible to do useful processing of partitionable operators on the accessible values.

The reader may have noticed that we used b_i to denote the element *of a multiset* as well as the element *of a partition of a multiset*. The reason is that there is no difference between the two as far as the applicability of partitionable operators is concerned. When an operator is applied to a multiset of elements, each element of the multiset is acted upon by the operator. Suppose that the multiset is singleton (i.e., the multiset contains a single non-duplicated element). In that case, the result of applying an operator to the multiset is another singleton multiset. The new multiset contains an element which is the result of applying the operator to the element of the original singleton multiset. Thus, for the purposes of applying an operator to a multiset of elements, the multiset can be equivalently considered to be partitioned into singleton multisets, each containing one element. We shall, henceforth, do so in order to make the presentation clearer.

Let a data item d be such that $\Pi^{-1}(d)$ is partitioned into multisets $b_1, b_2, \ldots, b_m$. Without loss of generality, let $b_1 = x_1, x_2, \ldots, x_p$. Let g be a partitionable operator in this context, that can be effectively applied to x_1. We then have the following :

$\Pi(g(x_1), x_2, \ldots, x_p)$: The value represented by the multiset $\{g(x_1), x_2, \ldots, x_p\}$
$= g(\Pi(x_1, x_2, \ldots, x_p))$: g applied to the value represented by the multiset $\{x_1, x_2, \ldots, x_p\}$
$= g(\Pi(b_1))$: Note that b_1 is the same as the multiset $\{x_1, x_2, \ldots, x_p\}$

Also, we can apply the same operator g to a multiset of elements where each element is a singleton multiset as described above, resulting in the following:

$\Pi(g(\Pi(b_1)), \Pi(b_2), \ldots, \Pi(b_m))$
$= g(\Pi(\Pi(b_1), \Pi(b_2), \ldots, \Pi(b_m)))$
$= g(\Pi(\Pi^{-1}(d)))$
$= g(d)$

In essence, a partitionable operator can be safely applied to any multiset of values that is a portion of $\Pi^{-1}(d)$, thereby deriving its obvious usefulness in a partitioned situation. That is, the operator may be applied to the values that are accessible.

Another facet of partitionable operators is that it is easily demonstrated that two such operators can be simultaneously applied to separate portions of $\Pi^{-1}(d)$ and that, in such situations, $g(h(d)) = h(g(d))$ where g and h are partitionable operators associated with a given mapping Π. Thus we can safely apply any number of such operators to do useful work within partitions.

We note that the concept can be extended to operators that affect more than one data item at a time. In this paper, we shall be mainly concerned with transactions that use such operators.

4.2 Virtual Messages

We shall assume that there is an unbounded totally ordered sequence of unique message identifiers for communication from a site s_i to a site s_j, and each distinct message (that is a message not generated due

to re-transmissions) carries such an identifier. This requirement can be relaxed; it is used to make our presentation more uniform.

Assuming unique message tags that are in some order, and piggybacked acknowledgements, common schemes (e.g., 'window' protocols) used in computer network applications [10] may be used to ensure that the value of N_i reaches site s_j eventually (although the transaction requiring the value may have aborted due to the lack of a response by then). We note, however, that the value is not lost forever by using Vm. We say that a Vm comes into existence the moment a log record indicating a message dispatch from site s_i to site s_j is created. The Vm ceases to exist the moment a log record (at the recipient) is created that indicates its reception *and* suitable disposal (e.g., in the example of Section 3, N_j is incremented by the value in the received Vm). A Vm is never lost, although several real messages corresponding to it may be sent during its lifespan. Essentially, a Vm is a conceptual tool to ensure guaranteed delivery (it assumes that if the same message is sent often enough across a link, it will eventually be delivered).

Every message that is sent from site s_i to site s_j should carry a piggybacked acknowledgement which has a message identifier m. Such an acknowledgement indicates to the other site that all messages upto and including the message m have been received and processed safely (that is, all relevant log records regarding those have been written and the other site will never be requested to send them again).

Site s_j, upon receiving a message must use some suitable means to record that the appropriate messages have been acknowledged (and possibly send those that are still outstanding). In all cases, duplicated messages are, of course, discarded. Standard protocols to handle these aspects are available [10] and we shall henceforth presume their usage.

The creation of Vm messages is now examined. It may happen that several Vm messages need to be created at once, and several database changes also have to be made. Initially, all the database item-values that are accessed are locked and the changes that are required are computed. Following this, a sequence of these database changes — *database-actions*, and a sequence of messages that need to be sent — *message-sequence*, are created. For the sake of exposition assume that all the information required about the messages is recorded; that is, the replicas and destinations of the messages constitute the sequence. To create the Vm, these two sequences are written into the log as *one* record of the form:

[*database-actions, message-sequence*]

Now the real messages are sent, and the database is updated. Following this, the locks are released. To complete the lifespan of a Vm, first the locks are obtained on the database item-values that need to be changed. The changes that need to be reflected in them are computed and logged as :

[*database-actions*]

Following this, the changes are carried-out, followed by the release of the locks. At this point, the reader may compare the above descriptions with the example given in Section 3 to help fix the ideas.

The removal of a Vm is a redistribution operator on the concerned multiset of values representing the data item. Any number of such Vm may be safely created. Thus, if a message corresponding to a Vm arrives long after it is due, it is always possible to safely carry-out the required action as long as the item in question is not locked. If it is locked, the message can be ignored; it will eventually be sent again anyway.

It may be noted that the acknowledgement protocol and the handling of Virtual messages must be carefully coordinated.

Henceforth, unless specified otherwise, we shall assume that the mechanism described above is always used. To clarify the presentation, we shall specify the logging of messages corresponding to Vm only (although the scheme outlined does require that the other messages also be appropriately logged). Indeed, it is not essential for each message to have a unique identifier and this will become apparent further ahead. We use a mechanism with unique identifiers to make the description of the scheme simple.

5 Transaction Processing

In this section, we describe how transactions are executed in our system. Though the methods shown here are applicable to the case where several transactions run concurrently, we shall initially assume that the description to follow is for the case where only one single transaction is running in the system at a time. We shall consider the more general case where several transactions run concurrently in the next section. Thus, the locking is described here, while not being necessary for a single transaction running alone, it would be necessary in the case that several transactions are running concurrently.

We assume that each transaction has a unique identifier, and that messages corresponding to any transaction carry these identifiers as part of their contents. A transaction executes in two phases:

1. Redistribution of data occurs so that the relevant information is gathered at one site. Till that point, there is no change in the value of the various data items.

2. The required computation is completed and the various log records concerning writes are written into stable storage. Hence, the (local) database is updated to reflect the changes.

Consider a transaction t initiated at site s_i. Let the data items accessed by t be denoted $A(t)$. We assume that all locks obtained by transaction t are exclusive locks. The transaction executes in the following sequence.

1. For each $d \in A(t)$ lock the local value d_i. These locks are obtained atomically.

2. For each $d \in A(t)$ such that d_i is inadequate for successfully executing the transaction t, send requests for relevant elements d_j and start a timeout counter. For all items to be *read* (in the traditional sense), requests for 'read' are sent to all sites that may have a value d_j corresponding to such items d. For other items that need not be 'fully' read, requests corresponding to the values required are sent (i.e., requests are sent to only obtain a proper subset of $\Pi^{-1}(d)$). All such requests are to remote sites requesting that all or part of the values of the d_j stored there be sent to site s_i.

3. Await replies for the requests sent until the timeout counter signals — whereupon declare an abort and then release the locks. This step exemplifies the pessimism that we incorporate for the sake of exposition — a timeout always results in the abortion of the transaction.

4. When all necessary replies have arrived in the form of Vm, the items in question are read, the requisite computation is done, and consequent updates are made ready. The operators used to effect the computations are partitionable operators.

5. Write log records regarding necessary changes to the database. The completion of this step commits the transaction while the failure of site s_i before the completion of this step aborts the transaction.

6. Make the changes to the local database and then record on the log that the changes have been made. The fact that the changes have been recorded in the database is logged for the purposes of recovery mechanisms that are described in Section 7.

7. Release all locks.

At this point, the reader is again referred to the example in section 3 to get a clearer picture of how the system works.

Several observations are in order. The site executing transaction t takes the pessimistic approach that any action that is delayed due to either the concurrency control mechanism, or the non-availability of locks, results in the abortion of transaction t. There are variations to our scheme where such a drastic action is not required. For example, the requests could be re-tried a few more times. However, we shall not go into those details to keep the presentation straightforward. What exactly is done by a remote site upon obtaining a request from another site will be described later. Note that till the fifth step, no data item in the database changes its value. Till that time, the first stage of redistribution is in progress. Further, there is no concept of rollbacks since once the log record in the fifth step is written, the transaction is committed. Before that, the abortion of the transaction has no effect on the database. The use of partitionable operators allows us to perform the write operation at the single site itself. It should be clear that all the Vm directed out of site s_i, and all the Vm directed to site s_i that precede the requests and the resulting Vm, respectively, will have reached their destinations safely if the required Vm is accepted at site s_i for the transaction t.

We note that the acceptance of a Vm requires that the data item value pertaining to it should be correctly updated and the acknowledgement for the Vm must be sent. In an earlier discussion, it was indicated that the data item in question must be locked in order to carry-out the acceptance of the Vm. However, if a transaction has already locked the data item, it can perform the actions necessary for the acceptance by itself. In such a case, there is no need to wait for the lock to be released. The restriction was placed earlier to indicate that the arrival and change resulting thereby to the database should constitute one atomic action.

Let us now examine the operation of a remote site s_j when it receives a request for some local data value d_j. First, if there is currently a lock on d_j, site s_j can simply decide not to honor the request. In such a case, the effect is to just ignore the request except for acknowledging its reception. Now let us assume that site s_j decides to honor the request. If the request is for a *read* at site s_i, then if there are no outstanding Vm for that item at site s_j, a Vm of that value is formed (and the value of d_j, which is the data value stored at site s_j, is changed to reflect this) and sent to site s_i in the manner described in an earlier section. The fact that no outstanding Vm is there, assures that the complete $\Pi^{-1}(d)$ is procured by the requesting site. If the request is not specifically for a *read*, then outstanding Vm need not be considered. The log record showing the formation of the Vm must reflect that the message was received and acted upon and the

acknowledgement protocols should conform with this. It is also appropriate to point out that if a request for creating a Vm, as in the case of a read request, arrives late, it can always be acknowledged but otherwise ignored. The messages will never be accepted if they are out-of-order by the scheme that we mentioned earlier.

The example of section 3 can again be considered to see how the descriptions so far fit in.

At this point in time it should be evident that the protocol described above is non-blocking. There is no need to detect failures of any sort (as remarked earlier, 'malicious' failures are not under consideration). If a message arrives late, regardless of whether it is a request or one corresponding to a Vm, it is handled as follows. A request, if necessary, may always be ignored. This may be prompted by considerations such as knowing that the response to the request is of no use to the requesting site (for example, if it is known to have arrived too late), or if the value concerned is locked. A Vm may always be accepted. For data items whose value is not locked, it is definitely always possible to process Vm arriving in the manner described earlier. In the case that the data item is locked by a transaction t, the Vm is accepted by the transaction without requiring to acquire locks for the concerned data items at the start. It should be apparent now that by restricting the writes to one site, the non-blocking behavior is achieved. An *exclusive* write (i.e., a write in the traditional sense) of a data item a mechanism is required to obtain locks at all the sites. However, as can be expected, this can lead to blocking behavior and, hence, we assume that the transactions do not require such a mechanism since we are concerned mainly with partitionable operators.

In case of write-only transactions, the initial steps of data redistribution can be ignored to obtain the following :

1. Lock local items $d_i \in A(t)$.
2. Compute necessary changes.
3. Log the changes.
4. Make the changes in the database and record this fact.
5. Release all locks.

Lastly, let us consider how the redistribution of data values is handled by redistribution-only (henceforth referred to as Rds) transactions. The sole purpose of such transactions is to redistribute data values without changing the values of data items. We shall see in Section 6 that Rds transactions are conceptually useful for the purpose of understanding concurrency control mechanisms. Rds transactions may actually not redistribute any data item at all. As the following example demonstrates, Rds transactions may simply be used to send requests for some data values to other sites.

1. Lock local items $d_i \in A(t)$.
2. Send requests as necessary.
3. Release all locks.

Observe that there is no need for the transaction to await replies. This follows from the fact that the arrival of Vm can always be taken care of by using Rds transactions that accept Vm in place of sending requests. Similarly, we shall assume, for conceptual clarity, that transactions whose sole purpose is to create Vm are also used by the system. It is interesting to see that general transactions for the systems of interest here can be regarded as combinations of Rds transactions with a write-only transaction. This observation is related to the notion of nested transactions and this is discussed in Section 8.

Let us now consider how transactions handle cases where not all the values requested are accessible. All items that are accessible and whose Vm have arrived can be used for subsequent processing. For example, in the seat allocation scheme, a site may require some extra seats and may request several sites for some extra seats. Suppose that one of the other sites responds with a sufficient number of extra seats, then the inaccessibility of any further data values makes no difference to the successful execution of the transaction at the requesting site. It is immaterial whether the other values are inaccessible due to link or site failures. In the case that none of the responses to the requests reach the requesting site, or that the requesting site is unable to obtain enough seats, the transaction is aborted. When available, further information, such as the knowledge that the lack of a response is due to the failure of the other site, can be always incorporated. However, in this paper, we restrict attention to the case that such information is not available.

The role of partitionable operators is very visible here. Only as many values as needed to effectively apply the operator need be gathered. Hence, even without detecting a network partition occurrence (or, indeed, other kinds of failures), processing can be done. With regard to non-partitionable operators such as the ones which may be encountered in traditional databases, one can envisage how they may be handled. In case of *reads*, all the values must be gathered. In case of an *exclusive* write, some kind of locks must be placed at the different sites (which could lead to blocking behavior at those sites, as remarked earlier).

6 Concurrency Control Issues

In contrast to traditional database systems, the notion of *serializability* is not immediately clear in the context of our proposed scheme. We suggest conforming to the traditional approach of correctness by requiring that any concurrent execution of transactions should be equivalent to a serial one. We assume that each transaction is designed to run correctly in isolation. Let us examine this more carefully. As there is only one copy of every data item d, represented by the multiset $\Pi^{-1}(d)$, the notion of a replica control strategy is not germane. The notion of serializability that we shall impose is that, subject to redistribution, the effect of a concurrently executing set of transactions is equivalent to a serial schedule where each transaction runs in isolation. The redistribution factor is present because although a transaction may not affect the total value of a data item d, it may change $\Pi^{-1}(d)$.

Let us now examine what is meant by running a transaction in isolation. Consider a system where just one transaction is initiated and it runs to completion. Although it is complete, Vm and requests associated with it may be active — with no harm done as they simply affect the distribution and not the values of the data items. If another transaction starts *after* the first one is completed (in real time), the two transactions are said to have run serially and in isolation.

This notion is extended to the case where several transactions run serially, one after the other, such that the next one in the sequence is initiated after the one under consideration runs to completion. In such a serial execution of transactions, reading the whole value of a data item (i.e., reading of a data item in the traditional sense) can be correctly accomplished. This is because all sites other than the site where the read is performed will have null values and there will not be any Vm corresponding to the data item at the time the reading (of the local value) is actually done.

Considering serializability subject to redistribution of data values is reasonable. It is always possible to run Rds transactions serially with other transactions. In fact, we shall make use of the fact that we can allow Rds transactions to run *concurrently* with any transaction without affecting the integrity of our schemes. It should also be noted that aborted transactions can be regarded as Rds transactions since the data items are unaffected except for the possible redistribution of their constituent data values (see Section 5).

For the sake of exposition of the concurrency control schemes, we have two types of transactions:

- Real transactions, and
- Rds transactions.

A real transaction is one that may actually affect the value of a data item. Rds transactions, as we have seen, affect only the data values of a data item without actually changing the data item. Each transaction conceptually executes at a single site. Several Rds transactions may be associated with a single real transaction. We use Rds transactions in a conceptual manner. Assume that if a Vm is accepted at a site when no transaction has locked the data value that pertains to it, then an Rds transaction is considered to have acted at that site which actually accepted the Vm. Similarly, for the requests that are honored at a site, assume that requisite Rds transactions actually make the necessary changes on the data values and create the necessary Vm.

As we have seen earlier, the following model of execution for a transaction (i.e., a real transaction with the associated Rds transactions):

1. Obtain locks on necessary local data values.
2. Send requests to different sites, if necessary.
3. Upon receiving requisite responses, do necessary computation.
4. Perform necessary local updates and release locks.

This model of execution is essentially the same as that described for transactions in Section 5.

6.1 Concurrency Control using Timestamping

In the following we suggest a new concurrency control scheme which is based on timestamping techniques. The scheme ensures serializability as we have defined above. Although the scheme is not necessarily optimal, it demonstrates how one can guarantee that our correctness criterion is met.

Assume some standard unique time-stamping mechanism. Every transaction t is given a (unique) timestamp $TS(t)$ which also serves as its identifier. Every data value d_i has a timestamp $TS(d_i)$ associated with it that specifies the last transaction to have locked it.

We use a very simple (and conservative) concurrency control protocol which we shall refer to as Concl. Let t be a transaction that is initiated at site s_i. Assume that it sends a request to site s_j for a data value d_j. Site s_j will send a Vm with the requested value only if $TS(t) > TS(d_j)$ subject to the other criteria that we have already considered in earlier sections (such as the availability of resources etc.). Thus, transaction t can effect a lock on a data value d_j at site s_j only if $TS(t) > TS(d_j)$. This is true even for $i = j$. As described in earlier sections, if a minimum number of Vm do not arrive at the site that sends the requests, the transaction is aborted. We assume that the locking and updating of the timestamp for a data is done in one atomic step. Any required processing is carried out before the lock is released (as described in an earlier section). In summary, the locking is handled as follows.

```
if TS(t) > TS(d_j)
  then begin
            lock(d_j);
            TS(d_j) := TS(t);
            do the required computation;
            unlock(d_j).
       end
```

As we have assumed that 'writes' to the local databases are carried-out only after log records are written, and that they are restricted to a single site alone, there is no possibility of a roll-back as every transaction reads only committed values.

We now proceed to demonstrate that our concurrency control scheme ensures serializability. Consider a set of transactions T = $\{t_1, t_2, \ldots, t_m\}$ that were executed under Concl and produced an execution sequence R. Assume, without loss of generality, that the transaction timestamps are also in the same order (i.e., $TS(t_i) < TS(t_{i+1})$). Also assume that there are no messages active in the system before the first transaction in R runs. We shall construct a serial execution sequence R' that is equivalent to R. For each transaction t_i in R, we denote by t_i' the corresponding transaction in R'. We require that for every i, t_i and t_i' are essentially executed in the same way. Let the system be in the state that it was before the execution sequence R took place. Initiate t_1 in the system and let it run exactly as it did in R except for certain differences which we describe below. Consider the next transaction t_i in R, and initiate t_i' corresponding to it similarly, after transaction t_{i-1}' runs to completion in R'. Besides these transactions, certain other Rds transactions will be considered to run in R' as described below.

Any message that arrives at a site during the execution sequence R' is generated by a transaction with a timestamp value that is less than the one for the transaction that accepts the message (recall that we assume that all events are a result of some transaction by making conceptual use of Rds transactions). For messages that were lost in R, assume that the same happens to corresponding messages in R'. Note also that if a site failed during the execution of R, it can be handled in R' by assuming that the same site fails at opportune

times during the execution of R' so as to exhibit similar behavior (we postpone the discussion of recovery procedures to the next section).

Every transaction in R' runs in exactly the same manner as in R. Firstly, assume that the data values observed by every transaction in R' are the same as those observed by corresponding transactions in R (how this is achieved is described ahead). Transaction t'_i sends the same set of requests as t_i. Those requests in R that are able to obtain responses from other sites have corresponding responses in R'. We assume that requisite Rds transactions are run at different sites to emulate the actions that are effected by the requests originated by t'_i in R' to correspond to the actions that are effected by t_i in R.

By making use of Rds transactions, we shall ensure that every transaction t'_i observes data values which are the same values as the corresponding transaction t_i, except, perhaps, for the timestamps. Clearly, this is true for t'_1 by the initial conditions. Assume that it is true for t'_{i-1}. After t'_{i-1} runs to completion, we redistribute the data values using Rds transactions so that the values that are observed by t'_i are the same as those by t_i. In this regard, certain Vm could have been accepted by a transaction t_p in R that were generated as a result of the actions of some other transaction t_q. By the construction described for R', we have assumed that these Vm are accepted (by an Rds transaction) *before* t'_p is initiated in R'. Hence, before t'_p is initiated, another Rds transaction is run at the site where t'_p is to be initiated that sends a Vm to the *same* site such that the Vm arrives during the execution of t'_p in the same manner as a Vm arrived in R during the execution of t_p.

Clearly, the timestamp sequence of each d_i in the system is consistent with the order of the transactions in T, in both R and R'. Furthermore, all transactions in R' are essentially the same as those in R leading us to conclude that the serializability of the transactions has been maintained. Hence we have shown that the concurrency control scheme Concl ensures serializability.

6.2 Concurrency Control using Two-phase Locking

Under certain reasonable characteristics of the system, we can use two-phase locking to achieve concurrency if the transaction processing is modified slightly. The protocol requires strict two-phase locking with respect to the data stored locally at each site for every transaction (real or Rds) that executes at the site. For example, consider a site that receives some requests for data values from a particular transaction (which may even have originated at the same site). If the site decides to honor the requests, it locks the requisite data values using strict two-phase locking.

Let us examine the system characteristics that are sufficient to ensure that our locking scheme works. Firstly, the communication links should preserve message order synchronicity [4]. That is, if a site s_k receives some messages m_i and m_j from sites s_i and s_j, respectively, and message m_i arrives before m_j, then m_i was sent earlier, in real time, than m_j. In this context, if messages arrive at the same time at some site s, then s can decide unequivocally which one arrived first based upon a total order on the sites. We assume, quite reasonably, that messages that arrive at a site are processed in the order of their arrival. Finally, we require that the sites can broadcast messages. That is, a site can send a set of messages without failure during the sending.

The above assumptions imply the following useful property. Suppose that site s_i and site s_j broadcast

a set of messages each. If site s_k receives the message from s_j after it receives the message from s_i (from the sets of messages broadcast by s_j and s_i), then every other site receiving the messages from both the broadcasts will receive them in the same order.

Let us now return to the main issue of transaction processing. We require that all the requests that are made by a transaction are broadcast together, and that all transactions executing at a site have their requests broadcast in the same order as their initiation. As we see next, the above assumptions are sufficient to guarantee serializability if two-phase locking is followed. We shall use the term Conc2 to refer to the two-phase locking concurrency control scheme in a system with the above properties.

We proceed now to show that Conc2 ensures serializability. Impose a partial order on the transactions that execute in the system in the following manner. All transactions executing at one site are (totally) ordered by the sequence of their initiation. Transactions that execute at different sites are ordered as follows. If transaction t_i sends a request to site s_j and the request is honored, then a transaction t_k, initiated at s_j, is ordered before (after) t_i if it accesses the same data value pertaining to the request and the access is effected before (after) the request is honored. Due to the system characteristics assumed, all the transactions in the system are partially ordered, and each data value stored in the system is accessed in an order that is consistent with the partial order of the transactions.

Now consider (hypothetical) timestamps on the transactions that are consistent with the partial order described. Also consider similar timestamps on the data values in the order that they were accessed (which is, clearly, consistent with the timestamp order of the transactions). With these timestamps, observe that Conc2 is a special case of Conc1. The difference is that Conc2 obeys two-phase locking at each site. Hence, if we consider a set of transactions and their execution sequence just as just as we did for Conc1, we can use the same approach as used for Conc1. Thus, we conclude that given the a system with characteristics as mentioned above, the concurrency control scheme Conc2 ensures serializability.

7 Recovery

Our scheme permits a particularly simple recovery mechanism. We first describe the algorithm that needs to be executed when a site recovers. In what follows, we make the reasonable assumption that a site knows that it failed. The algorithm is stated simply:

1. Release all locks held by any transaction on local data values (if the locked state survives a failure).

2. Access log records to find committed transactions and redo all the changes to the local database.

Let s_i be a recovering site. Consider the data values d_i that were in a locked state when the failure occurred. All these locks can be safely released (this assumes that lock states survive the failure). In fact, this leads us to conclude that the information regarding the locks need not survive a failure in a site. The reasons should be clear. Consider a site s_i that had some items in its local storage locked due to requests originating from another site s_j. Suppose that site s_i failed while the locks were still in effect. Then the remote site would not receive any message unless the log records corresponding to the message, which is a Vm, had been created. If no Vm arrived at site s_j in time, the site would continue processing the transaction it

was involved in without the Vm, if possible. Otherwise, it would abort the transaction. If a local transaction had locked it, it would have been an abortive attempt (if the final log records for it were not available) or it would have committed. In either case it is safe to ignore the locks and assume that no locks are held on any d_i after the failure.

Regarding the timestamps of d_i, it should be noted that for a committed transaction, all the accessed d_i would have correctly updated timestamps. For data values that were accessed by partially completed transactions the timestamp values would be at most larger than the values they would have had if the transactions had not been run at all. But this poses no threat since the only effect would be to prevent some otherwise possible transaction to run in the future. Hence, no actions need to take place regarding the timestamps of the data items.

As far as updating the value of a data value d_i that is affected by committed transactions is concerned, the logs need to be accessed to determine which transactions committed (i.e., which transactions had their *database actions* logged) but did not update the database itself. Those updates need to be redone. The redoing actions must be *idempotent* [5] in view of the possibility of a failure during the recovery phase. Note that by using checkpointing mechanisms, the number of redo actions required can be reduced in the usual manner [5].

One other problem that may be envisaged is that the timestamping may get outdated as the local counters were halted. This will not cause a problem since, in any case, all local timestamps would be unique (by attaching the site identifier in the low order bits of a timestamp — a common scheme) and any transactions attempting to execute from the recovered site would fail (initially). However, the reception of any messages that originate from other sites *after* the failed site recovered and received by the recovered site, would 'bump-up' the counter for the timestamps [5]. Hence, the outdated timestamps only constitute a temporary problem.

Finally, outstanding Vm need not be sent again. The guaranteed message delivery scheme for Vm that we have described in an earlier section ensures that the unsent Vm do not constitute a problem. The system eventually sends the outstanding Vm in the normal course of processing.

From the above discussions, it is evident that the recovery can be truly claimed to be independent. That is, other sites need not be queried to find out any information to allow normal processing to begin (we shall ignore the temporary time-stamping problem as it is clear that write-only transactions could always be processed at the local site). In particular, even if all sites fail and subsequently one site recovers, we have the case that it can begin doing some useful work (it is easy to see that the outdated timestamps will not affect this).

With regard to link failures, no special actions need to be taken. The links may lose, delay, duplicate messages or just fail. The protocol for message handling that we have used, and the timeout counters being used in the transaction processing assure that the system is not adversely affected by such behavior.

8 Discussions

As with any newly proposed scheme, one must demonstrate that the idea has some real merit. To do so, we include in this section a consideration of the apparent advantages and disadvantages of DvP and Vm.

Let us first examine some of the advantages of our newly proposed scheme. As outlined in the introduction, the problem of correctly managing data in a failure-prone environment is difficult. Our scheme has the advantage of simplicity. Its other useful properties include the non-blocking nature of its transaction processing, the lack of necessity to detect different kinds of failures, and the ability to recover independently. Furthermore, in the case of network partitions, there is still the possibility of continuing with the normal operations, thereby allowing quite high accessibility. The scheme is deadlock-free since there is no situation where an indefinite amount of waiting is involved. The correctness criterion that we have outlined is based on the acceptable notion of serializability which can be ensured by timestamping techniques. If some additional assumptions are made about the system, one can ensure correctness by using a simple locking technique. If the requirement of serializability is relaxed, additional advantages could be accrued. There are other minor advantages which we do not discuss here.

Where can DvP and Vm be applied profitably? Clearly, applications such as inventory control, airline reservations and banking applications can benefit. In such cases, the data and the operators are both naturally partitionable. The concept of Vm can be profitably used more generally. To ensure fault-tolerant data transmission, such schemes can be used to send crucial information from one site to another. An example of such use is in distributed databases with non-replicated (or with a very small degree of replication) of data. For example, consider a system for the electronic transfer of monetary funds. Messages in such systems entail information that should not be lost in transit. Suppose that the system has the characteristic that a message that is resent often enough will eventually reach the intended destination. In that case, the concept of Vm can be used to ensure that no message is completely lost and every message is delivered eventually. Thus, the information contained in any message is not lost by the system.

There are several situations other than traditional distributed databases where DvP could prove useful. Consider the applications where it is not possible to carry-out processing to ascertain the causes of failure, or where it is imperative that contention for certain types of resources be minimized. For example, a possible application where it may be interesting to consider the use of DvP is in systems that use 'aggregate fields' ([7] and references cited therein). These fields represent quantities where the type of update required is limited to increments or decrements. Very often, special concurrency control schemes are used specifically for such quantities since they are accessed very frequently and can become 'hot spots'. In such a situation, using DvP may alleviate the problem of contention by allowing several processes to access a particular quantity simultaneously. While this should work well until a *read* (in the traditional sense) is required, in the case that several of the data-values need to be accessed, it may be preferable to resort to traditional schemes. To make the best of both approaches, it may be preferable to design systems that can respond to different situations by dynamically interchanging between a DvP scheme and some traditional scheme.

Our scheme cannot, in its current form, be applied to all database applications. However, as we remarked, it can be effectively used in many types of applications. Although there is a high overhead in reading the entire value of a particular data item, it should be noted that in traditional environments in case of different

failures, if updates of data items were allowed elsewhere, it would not be possible to read at all. In our scheme, the failure of a site implies the inaccessibility of the information exclusively residing there; but in a traditional non-replicated database, this is the case as well. In a replicated database it would not be possible to distinguish between site failures and network partitions in which case it would be unsafe to access the information, except to read, at all locations. Even the read access would not be permitted if updates to the same data item are allowed elsewhere. Finally, it is likely that only a small amount of the resource represented by the value stored at the failed site is inaccessible. The rest is accessible. There is a problem of livelock occurring in the scheme as described, but using some additional mechanisms, this can be avoided.

In this paper, we have not addressed the issues of performance. Such a study would involve the examination of different applications, the data types, the transactions required, network topologies, processing capabilities and several other issues. To use the new approach well it is necessary to study performance aspects, but they lie beyond the scope of this paper.

The case for DvP and Vm can be viewed in a slightly different light. Significant effort and research has been spent in trying to make failure-resilient systems. However, the results do not appear to be commensurate with the efforts. The reason is, of course, that it is a difficult problem. Perhaps the answer lies in radically different approaches to the solution of the problems. This paper should be viewed as a step in this direction.

It should be pointed out that several optimizations are possible to improve the efficiency of the system. We could introduce different locking modes; allow more optimism in the protocol for concurrency; not insist on unique identifiers for all messages (e.g., request messages need not have unique identifiers as their delivery is not critical); etc.. Indeed, studying the properties of partitionable operators and representations could yield a host of other optimizations. Also, if the system has more favorable synchronism characteristics, it may be possible to find better concurrency control schemes. For more general systems, some hybrid of DvP and traditional approaches could also be considered.

Another improvement that may be explored, as remarked earlier, is to use nested transaction techniques [6] to increase parallelism in the execution of the transactions. Notice that the transactions may require to read and write several data items and the earlier description of using several Rds transactions is easily seen to be an application of nested transaction methods. By using nested transactions, the parallel execution of transactions, even at the local sites can be increased.

Finally, we note that there has been a previous suggestion to use the split-up value of a data-item to tolerate network partitions [3]. The suggested approach handles data in a partitioned manner only during a network partition failure and is, therefore, very different from our approach which considers the data partitioned at all times. Also, the notions of consistency and serializability that we use are obviously not used elsewhere. Furthermore, in other approaches there is a need to detect the occurrence of partition failures that is not necessary in our scheme. We are not aware of any method that uses a split-up item handled in the manner outlined in this paper.

9 Conclusions

We have described a new scheme for storing and handling data in a distributed environment that is failure prone. The scheme has the major advantages of non-blocking behavior, high availability, simplicity, independent recovery, and does not require failure detection protocols. It may be worth studying the ideas further especially in the light of several related impossibility results such as the ones that exist for commit protocols.

We have presented an approach to handle a difficult problem. Two important aspects of the scheme need to be examined further. Firstly, there is a need to find ways to extend the methods to handle more data types. Secondly, performance studies to find the best ways to distribute the data, to design the transactions and to reduce the message traffic are needed.

Acknowledgement : The authors wish to acknowledge the incisive and helpful comments provided by Hank Korth on an earlier draft of this paper.

References

[1] Bernstein, P.A., and Goodman, N. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv. 13,2* (June), 185-221.

[2] Ceri, S., and Pelagatti, G. 1985. Distributed Databases (Principles and Systems). McGraw Hill Comp. Sci. Series.

[3] Davidson, S.B., *et al.* 1985. Consistency in Partitioned Networks. *ACM Comput. Surv. 17,3* (Sep), 341-370.

[4] Dolev, D., *et al.* 1987. On the Minimal Synchronism Needed for Distributed Consensus. *JACM. Vol.34, No.1* (Jan), 77-97.

[5] Korth, H.F., and Silberschatz, A. 1986. Database System Concepts. McGraw Hill Adv. Comp. Sci. Series.

[6] Moss, J.E.B. 1986. An Introduction to Nested Transactions. *COINS Technical Report 86-41* (Sep).

[7] O'Neil, P.E. 1986. The Escrow Transactional Method. *ACM TODS Vol.11, No.4* (Dec), 405-430.

[8] Skeen, D. 1982. Crash Recovery in a Distributed Database System. Doctoral Dissertation, Dept. of Elec. Engin. and Comp. Sci., Univ. of Calif., Berkeley (May).

[9] Soparkar, N.R., and Silberschatz, A. 1989. Distributed Systems are Indecisive. *Submitted for Publication.*

[10] Tanenbaum, A.S. 1981. Computer Networks. Prentice-Hall, Inc., Englewood Cliffs, N.J.