Abstract

Network Partition failures in traditional Distributed Databases cause severe problems for transaction processing. 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 the techniques could be profitably implemented in a distributed environment.

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 sites fail, or the communication links 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 [9, 10].

Network partitions result when a system of sites connected by communication links get separated into more than one group such that different groups are unable to communicate. This may be caused by the failure of the communication links, the sites, or both. The difficulty encountered in such situations is two-fold. First, transactions in progress when the failure occurs may be blocked as they are unable to commit or to abort safely [3, 9]. This results in the non-availability of data held by these transactions. Second, if data is not replicated, then after the network has partitioned, 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. In case of replicated data, after the network has partitioned, it is unsafe to allow changes to be made to locally available copies as the changes are not available at all the sites.

This scenario becomes more complicated in the situation of several partitions occurring at the same time, the network getting restored, and recovery processes executing in a dynamic environment. In fact, the concept of a network partition itself may become difficult to define. The meaning of a partition that is not "clean" is unclear [3], and the meaning of detection of failures is also imprecise. In such a situation, long delays in communication incurred between two groups of sites becomes characteristics of a network partition [10].

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. The new approach embodies certain very interesting characteristics. It permits a uniform approach to handling the different kinds of failures without the need to detect them, or to ascertain their type. The methods allow the handling of an arbitrary number of site failures that may occur at any time. The recovery mechanism in case of site failures is particularly simple. Also, we are able to handle all cases of link failures that include the loss, duplication, delay, and re-ordering of messages, and we make only minimal progress assumptions about the link behavior. The concurrency control
2 Problems Associated with Network Partitions

We assume that there are \( n \) sites that 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 [9]. 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 a given 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, as long as the failures that occur are from the given set [9, 10]. A commit protocol is said to be blocking if it is not non-blocking. A transaction making use of a protocol, is said to block if it is unable 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 transaction is awaiting the occurrence of some event such as the reception of a message from another site.

The non-existence of a commit protocol in several key situations is well established. Let us assume that the network partition failures occur in the system in only one specific manner. Assume that only the following restricted type of network partition failures (which we call well-behaved partitions) may occur in the system.

1. Only a fixed set of links may fail to form non-communicating groups of sites.

2. Partitions form when a set of the links fail simultaneously so that their failure produces at least one more isolated group.

3. When a link does fail, its failure results in the isolation of one of the groups (i.e., if any one of them did not fail, the number of non-communicating groups formed would be fewer).

Theorem 1 [9, 10]: There exists no commit protocol to handle the case where well-behaved partitions may occur in a system with three interconnected sites. Furthermore, there exists no commit protocol to handle the case of well-behaved network partitions that form more than two non-communicating groups.

Now, consider the related question of network partition detection. Several existing methods to handle failures assume that it is possible to detect network partitions. However, in case both the links as well as the sites are failure-prone, this is not a correct assumption to make. This is because the only practical methods used to detect failures use time-out mechanisms. That is, when a certain event, such as the reception of an expected message, does not occur within a pre-specified time limit, then a failure of some sort is said to have been detected. Consider two failure-prone sites \( s_i \) and \( s_j \) connected by a failure-prone communication link. If a message from site \( s_j \) that was expected to arrive at site \( s_i \) fails to arrive, then site \( s_i \) is able to assert only that a failure occurred in either the link, or the site \( s_j \), or in both. Note that in this situation, a link failure is a network partition whereas the failure of site \( s_j \) alone is not. Thus, in general, it is not possible to detect network partitions [10].

The implications of these results 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 should be several sites at which updates need to be made. This is because it may be possible to avoid blocking due to the lack of communication with the sites where updates are not necessary (i.e., sites that have read-only data). For example, in a lock-based concurrency control scheme, it may be possible to release the read locks on the data by some time-out mechanism. However, updates must be recorded atomically. Other methods that involve replicated data, and which do not require that all the writes occur atomically, need techniques similar to having a majority of the sites implementing the write — which leads to the same problem. The blocking behavior follows due to the requirement that a transaction either commits or aborts atomically across all locations [2]. Hence, we have the following result.

Theorem 2: Transaction processing could be guaranteed to exhibit non-blocking behavior only if each transaction is restricted to writing data at one single site.

3 The New Approach

The above discussions suggest that a new approach to distributed transaction management must be developed if one wishes to employ non-blocking protocols. 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. In this section, we illustrate our main ideas using examples.
Consider a data item denoting the number of vacant seats in an airline reservation system. We can split the number into several smaller portions. Each of these portions itself denotes a number of seats. 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 a number of seats. Notice that we regard the data in a manner similar to the way in which resources are regarded in an operating system.

Let us consider the example in more detail. Consider a simple airline reservation system, where the system reserves seats for a flight “A”. Assume that seats can be reserved from four sites $s_1$, $s_2$, $s_3$, and $s_4$. 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 a flight $A$. In our system, the sites $s_1$, $s_2$, $s_3$, and $s_4$ store values $N_1$, $N_2$, $N_3$ and $N_4$, respectively, in connection with $N$. The relation between these is $N_1 + N_2 + N_3 + N_4 = N$.

Initially, let $N = 100$ and $N_1 = N_2 = N_3 = N_4 = 25$. Note that $N$ is not represented explicitly at any site.

Consider the allocation of seats to the customers. If customers requesting 3, 4 and 5 seats arrive at site $s_1$ and their requirements are met, the values assumed by $N_1$ are 22, 18 and 13, respectively. These changes in the local value of $N_1$ 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 $s_1$, then the number of seats that become available as a result of the cancellation are added to $N_1$. Note that in doing so, it could turn out that $N_1$ exceeds the initial value of 25 that it had since customers that reserved seats at some other site could cancel their seats at site $s_1$. Thus far, the 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: $N_1 = 2$, $N_2 = 3$, $N_3 = 10$ and $N_4 = 15$. 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 $s_2$. Clearly, the request cannot be granted without increasing the value of $N_2$. The site $s_2$ decides to request seats from some other sites. Depending on how the system is function, a request for at least three seats is sent by site $s_2$ to one or more sites among $s_1$, $s_3$ and $s_4$. Once such a request has been initiated, a time-out mechanism at site $s_2$ is started and site $s_2$ awaits responses from the other sites. The motivation for sending the requests is to redistribute the value of $N$ among the sites so that the value of $N_2$ 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 $s_4$ receives the request sent by site $s_2$. Suppose that site $s_4$ decides to send 5 seats as a response. Site $s_4$ will decrement $N_4$ by 5 and send a message carrying a value of 5 to site $s_2$. 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 $N_4$, and a copy of the message sent to site $s_2$ must be recorded on a log residing in stable storage. Thus, in our example, the value of $N_4$ is reduced to 10 after a log record indicating this change and a message of value 5 to site $s_2$ is recorded on stable storage.

We assume that messages are numbered and that the failure of a timely acknowledgement from the intended recipient prompts re-sending of the message. A more complete description of this scheme, which we refer to as Virtual Messages, is provided ahead. By this mechanism, the data is sent by site $s_4$ to site $s_2$ safely. That is, the data is not lost as a result of failures in the system.

If all goes well and site $s_2$ receives the message from site $s_4$ (and, perhaps, from other sites as well), the value of $N_2$ is incremented by the amount of the data carried in the messages. For example, if site $s_2$ received only the message from site $s_4$, it increments $N_2$ by 5. Clearly, the receipt and incrementing of the messages is done in a careful manner that involves logging of relevant data — the details are provided in subsequent discussions. Once $N_2$ is correctly updated, the request for 5 seats by the customer is handled in the normal manner.

Consider the situation at site $s_2$ in the case that a time-out is signaled prior to the arrival of messages that make $N_2$ 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. There may be several reasons why a sufficient number of messages fail to arrive. For example, the other sites may have failed, messages may have been delayed, communication links may have failed, or there may not be any more available seats. What the reasons exactly are is immaterial. We do not require site $s_2$ (or any other site) to attempt any detection of the type or the correction of such failures external to the site.

Note that $N \geq N_1 + N_2 + N_3 + N_4$ at all times. Also, if $N_M$ represents the values in transmission, $N = N_1 + N_2 + N_3 + N_4 + N_M$. Thus, the need for a robust scheme of message transmission is not difficult to see. Also, if it is necessary to determine the value of $N$
at a site, say site $s_2$, then it must be ensured that $N_1 = N_3 = N_4 = N_M = 0$. In that case, $N = N_2$ and the value may be determined easily.

Let us summarize the above example. 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 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. In case of network partitions, each site is able to access at least its local data and also any data that is available in the sites that are accessible. Clearly, this is better than a situation where no processing is possible.

Suppose that site $s_i$ needs to send $N_i$ to site $s_j$. 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 now be removed.

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). Note that, in airline reservation systems, it is very desirable to have at least the allocation schemes that are non-blocking (e.g., to avoid losing potential customers who may be impatient).

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. 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 accessing only locally stored data, and making infrequent requests to other sites in the case of being unable to proceed with what is locally available. If the responses from the remote sites fail to arrive for any reason within a specified time-out period, the transaction is aborted. In order to access a data-value, a lock must be requested and obtained. 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 specific method for communicating data. By making use of stable logging facilities in conjunction with sliding-window protocols [11], it is ensured that data that is transferred is not lost. A more elaborate description of these methods is provided in the next section.

4 System Structure

We are in a position now to formalize our basic ideas. 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. At times in the following description, we shall sacrifice mathematical precision for the sake of a more clear exposition of the ideas.

4.1 Data-value Partitioning

Let $d$ be a data item drawn from a domain $\Gamma$. Consider a non-empty multiset, $b$, (i.e., a set with possibly duplicated elements) consisting of values drawn from $\Gamma$. The domain of the multisets is denoted by $\Gamma^+$. We say that a surjective mapping $\Pi : \Gamma^+ \rightarrow \Gamma$ denotes a Data-value Partitioning (DvP) of the elements of $\Gamma$. We use $\Pi^{-1}(d)$ to denote the multiset $b$ representing the partitioned values of $d$ where $d \in \Gamma$ and is stored in the system at any given time in the form of multiset $b \in \Gamma^+$.

The DvP need to have some additional properties. The mapping $\Pi$ 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 $\Pi(b') = \Pi(b)$. While the description is not mathematically precise, the intent should be clear.

Now we describe the type of operators with which we are primarily concerned. We say that $f$ is a partition-
able operator for \( \Gamma \) and \( \Pi \) if its effective application to an element \( b \) 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”. In that case, \( f(\Pi(b)) \) would not equal \( \Pi(b') \) since \( b' \) would be the same as \( b \). Examples of partitionable 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 partitionable operators that can be handled neatly in a non-blocking manner. We are essentially concerned 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. Each element of the multiset corresponding to a data item represents a data-value of that data item. Operators that cause such a change are 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 \( \Pi \) 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 could be easily found 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 with the accessible values using partitionable operators.

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 for this is that the above description implies that there is no difference between the two as far as the applicability of partitionable operators is concerned. 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.

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 that can be effectively applied to \( x_1 \). We then have the following:

\[
\Pi(g(x_1), x_2, \ldots, x_p)
\]  
(Value of \( \{g(x_1), x_2, \ldots, x_p\} \))

\[
g(\Pi(x_1, x_2, \ldots, x_p))
\]

\[
g(\Pi(b_1))
\]

When \( (b_1 \) is the same as \( \{x_1, x_2, \ldots, x_p\} \)

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:

\[
\Pi(g(\Pi(b_1)), \Pi(b_2), \ldots, \Pi(b_m))
\]

\[
g(\Pi(\Pi^{-1}(d)))
\]

\[
g(d)
\]

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 property that partitionable operators require is that two such operators, say \( g \) and \( h \), should be simultaneously applicable to separate portions of \( \Pi^{-1}(d) \), and in such a situation, \( g(h(d)) = h(g(d)) \). Thus we can safely apply any number of such operators to do useful work within the partitions. Clearly, the concept of partitionable operators may be extended to operators that affect more than one data item at a time. Also, we may associate a null value with an element of \( \Gamma \) which is regarded by every operator as the identity element value for that operator.

### 4.2 Virtual Messages

We 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 retransmissions, carries such an identifier.

Assuming ordered unique message tags, and acknowledgements, common “sliding-window” protocols used in computer network applications [11] may be used to ensure that a message destined for a location eventually arrives there. All messages in the system follow this protocol. The novel idea that we introduce is to incorporate the storage of data in a message — i.e., the data-value carried in a Virtual Message (Vm) is not reflected in the database (though it is recorded on stable storage for the message protocol). The value destined for a site reaches it inspite of link or site failures — although the transaction requiring the value may have aborted due to the lack of a response by then. Moreover, there is effectively an exclusive lock on the data-value held by the transaction receiving the Vm for
the duration of the existence of the Vm transporting the data-value since that data-value is not reflected in the database.

We say that a Vm comes into existence the moment a log record indicating a message dispatch from site si to site sj is created. The Vm ceases to exist the moment a log record is created at the recipient site that indicates the reception and suitable disposal of the 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. The use of a sliding-window protocol also guarantees that the messages are not delivered out-of-order, nor are they duplicated. We assume that a message sub-system of this nature is provided as a lower layer service to the database management system.

**Theorem 3:** All link failures in the form of lost, duplicated, reordered, and delayed messages, can be reduced to delay failures. □

Now, we briefly describe a standard message protocol, and also how we incorporate our ideas of Vm into it. Every message that is sent from site si to site sj should carry a piggy-backed acknowledgement which has a message identifier m. Such an acknowledgement indicates to site sj that all messages up to and including the message m have been received and processed safely. That is, all relevant log records regarding those messages have been written at site si and site sj will never be requested to send them again. Thus, site sj, upon receiving a message, must use some suitable means to record that the appropriate messages have been acknowledged, and re-send those that are still outstanding.

The creation of Vm 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. To do so, firstly, the data-values that are accessed at the site where the Vm are created are locked, and the changes that are required are computed. Following this, a sequence of these database changes, the **database-actions**, and a sequence of messages that need to be sent, the **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 the 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 data-values that need to be changed at the receiving site. The changes that need to be reflected in them are computed and logged as:


```
[database-actions]
```

Following this, the changes are made in the database, and then the locks are released.

The creation and the removal of a Vm are redistribution operators on the concerned multiset of values representing a data item. Any number of such Vm may be safely created. Thus, if a message corresponding to a Vm arrives delayed, it is always possible to safely do the required actions 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. Henceforth, we assume that the mechanism described above is always used.

**Theorem 4:** There exists a robust scheme to send data-values between sites which only incurs delay failures in transit. □

## 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 initially assume that the description to follow is for the case where one single transaction is running in the system at any given time. We consider the more general case where several transactions run concurrently afterwards. Thus, the locking that is described here, while not being necessary for a single transaction running alone, would be necessary in the case that several transactions run concurrently.

We assume that each transaction has a unique identifier, and that messages corresponding to any transaction carry the identifier as part of their contents. A transaction executes in two phases (the first is optional, and we conjecture that it occurs rarely in the systems of interest):

1. Redistribution of data occurs so that the relevant information is gathered at one site. Till that point, there is no change in the values 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 si. Let the data items accessed by t be denoted A(t). We assume, conservatively, 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. Common deadlock prevention schemes may be employed here.

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. If the items are 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 \( d \) using a protocol explained further ahead. If the items need not be "fully" read, requests corresponding to the values required are sent (i.e., requests are sent to obtain only 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 the site 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.

Several points need to be noted here. The transaction \( t \) clearly follows the Strict Two-Phase Locking protocol (Strict 2PL) with respect to the data values at the local site. 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 \( t \). What is done by a remote site upon obtaining a request from another site is described ahead. Note that until step 5, no data item in the database changes its value. Till that time, the first phase of redistribution is in progress. Further, in case of transaction aborts, no rollbacks occur since the Strict 2PL is followed and only committed values are provided to other transactions. The use of partitionable operators allows us to perform the write operation at the single site itself.

Since messages are permitted to be delayed, Vm carrying data-values pertaining to other transactions may arrive at the site while transaction \( t \) is still in progress. Transaction \( t \) must be so designed that if the Vm that arrive pertain to a data-value in \( A(t) \), and they arrive prior to the reading of values in step 4, then the transaction accepts and processes the Vm. We note that the acceptance of a Vm requires that the data item value pertaining to it should be correctly updated. In an earlier discussion, it was indicated that the data item in question must be locked in order to accept the Vm. However, if a transaction has already locked the data item, it can perform the actions necessary for the acceptance. 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 the 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 \( d_j \). Firstly, if there is currently a lock on \( d_j \) or the value of \( d_j \) is inadequate, site \( s_j \) can simply decide not to honor the request. In such a case, a Vm with a null value could be sent to the requesting site. If the request is not specifically for a traditional read of a data-item, the requisite Vm is created and sent to the requesting site. In all cases, the receipt of the request and the creation of the requisite Vm must be effected atomically by the use of log records on stable storage. Atomic execution of operations that are local to a site is not difficult since we are able to make use of the stable storage locally.

If a transaction requires the full value of a data item, then it must follow a protocol to gather all the data-values pertaining to that data item at its local site. Hence, a protocol similar to a global snapshot is executed where the pertinent data-values from all other sites are gathered, and all the communicating links are flushed of their Vm. This ensures that the data-value at the site that is performing the traditional read is equal to the value of the entire data item. The details will be provided in the full paper.

At this point in time it should be evident that the transaction management above is non-blocking. There is no need to detect failures of any sort ("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 Vm may always be accepted as long as it does not arise between the reading of data-values in step 4 and step 7 of a transaction which has the related data-value locked. If a Vm does arrive at such a time, it may be
safely ignored since it will be re-sent in any case. For data items whose value is not locked, it is always possible to process Vm arriving in the manner described earlier. Request messages may always be responded to with a null-valued Vm — unless the requests pertain to a traditional read. It should be apparent now that by restricting the writes to one site, the non-blocking behavior desired is achieved.

Lastly, let us consider how the redistribution of data values is handled by redistribution-only transactions (referred to as RD transactions). These transactions may be regarded as a conceptual tool for the uniform treatment of the Vm. The sole purpose of such transactions is to redistribute data values without changing the values of data items. RD transactions can be regarded as transactions that process the Vm, their sending with RD-send, and receipt with RD-receive. The RD transactions lock the requisite data-value, create or accept a Vm (and effect the requisite change in the database), and finally release the lock. All Vm have a pair of RD transactions for their creation and receipt — except in cases where the receipt is handled by ordinary transactions.

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. 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 effect the requisite redistribution, the transaction is aborted. 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 when available. However, in this paper, we restrict attention to the case that such information is not available.

The role of partitionable operators is very evident. 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 may continue. With regard to non-partitionable operators such as the ones which may be encountered in traditional databases, one can envisage how they need to be handled. In such situations, all the values must be gathered at one site.

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. Note that since there is only one copy of every data item d represented by the multiset II_1(d), the notion of a replica control strategy is not germane. From the earlier description in Section 5 regarding the manner in which a transaction is executed, notice that all the changes made by any transaction to the data items are effected in a single atomic action. Thus, aborted or partially executed transactions are equivalent to certain RD transactions. Hence, such transactions need not be examined separately. As mentioned earlier, we are interested primarily in partitionable operators and the description pertains to them. In what follows, the actions of a transaction denote the atomic operations that constitute the transaction. From the earlier discussion, the processing of the requests, the creation of Vm, and the acceptance of Vm may also be regarded as atomic actions.

Definition 1: An execution of the sequence of transactions t1, t2, . . . , tn is serial if for i < j, all actions of t_j occur after, in real time, the actions of t_i. □

What is really sought in the above definition is that the actions of transactions that occur later in a serial sequence should not causally affect the actions of those before. Hence, to formalize causality, we use the following relation “→” adapted from [5].

Definition 2: For two events a and b, a → b iff one of the following holds.

• a and b are events at the same site and a precedes b.

• a represents the sending of a message whose reception is represented by b. □

The transitive closure “→+” of “→” provides the causality relation among the events. Thus, for a serial execution in the definition above, for i < j, if a_i ∈ t_i and a_j ∈ t_j, then a_j →+ a_i.

Let us examine more closely the isolated execution of a transaction. Delays in message transmission mean that Vm and requests pertaining to other transactions may be active at the time that a transaction t_i runs. The way these are handled was described in Section 5. Moreover, the messages produced by transaction t_i may remain active long after the transaction t_i itself has completed. Thus, although a transaction may run in an isolated manner (in the sense that it is the only one active in the system in real time), it must be designed to deal with delayed messages that arrive while it is in execution. The serial order described is too restrictive. Actions causally affect one another only if they affect or access common data, or between them there is a sequence
of actions from the "→" relation such that adjacent actions in the sequence affect or access common data. That is, we are interested in the notion of conflicting actions. This is made more precise as follows.

Consider the multiset of the data-values that constitutes a data item. At any point in time, we can consider the augmented multiset for the data item to consist of data-values at the sites and all the Vm pertaining to the data item that were created in the system up to that point. This is possible if we regard all the elements that correspond to inactive Vm to have a null value, and every element that corresponds to an active Vm to have the value of that Vm. That is, the "life" of an element that corresponds to a Vm would reveal that it has a null value up to the point when the Vm pertaining to it is created, then it has the same value as the Vm up to the point when the Vm is processed at its destination — whereupon the element gets the null value again.

Now we are in a position to describe conflicting actions. Given that we are concerned with partitionable operators, actions that conflict are those that access or affect common data-values in the augmented multisets of the data items. Thus, we define the following relation "→" based on "→" and the conflicts among actions.

**Definition 3:** Let a and b be two actions that belong to some transactions. We define a → b iff

- a → b, and

- a and b conflict, or a and b belong to the same transaction.

We now examine certain issues related to the augmented multisets more closely. An RD-send transaction at a site si that creates a Vm for a data item d, conceptually locks both, the data-value di at site si and the data-value pertaining to the Vm created in the augmented multiset for d. After the creation of the Vm, the RD-send may be regarded to release the locks on both the data-values. Similarly, the transaction that accepts the Vm conceptually locks the data-value of the Vm sometime after the "lock" on it is "released" by the RD-send transaction. Note that aborted transactions are equivalent to a set of RD-received transactions that accept the same set of Vms as the aborted transaction did. Since the Vm is never accessed by any other transaction, it is safe to assume that the "lock" on it is "released" by the recipient transaction along with the release of its other locks. Thus, all transactions in the system may be regarded as obeying Strict 2PL with respect to the data-values in the augmented multisets.

At this point we draw attention to the notion of a history in traditional systems [1], and to the partial order "→" defined above that parallels it. The idea of the equivalence of histories and a serialization graph for the transactions may also be developed in a similar manner. Finally, the observation that the system obeys Strict 2PL on the data-values of the augmented multisets of data items allows us to use the techniques developed in the traditional theory to obtain the following.

**Theorem 5:** The Strict 2PL protocol followed by the transactions at each individual site is sufficient to produce serializable schedules in the system. □

### 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 lock states survive a failure).

2. Access log records to find committed transactions and re-do all the changes to the local database.

Note that in the above algorithm the recovering site does not require communication with the other sites. Hence, it is an independent recovery scheme. The reason that we are able to obtain a simple algorithm is because partially executed transactions in our system correspond to RD transactions. Let us examine further details. Let si be a recovering site. Consider the data values di that were in a locked state when the failure occurred. If the lock states survive a site failure, all these locks can be safely released. Consider a site sj that had some items in its local storage locked due to requests originating from another site sj. Since Vm formation is atomically effected, this does not constitute a problem. If no Vm arrived at site sj 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 a data-value, the transaction would be either be uncommitted (i.e., the final log records for the transaction were not written), or it would be committed. In the former case, the transaction may be assumed have aborted. In either case it is safe to ignore the locks and assume that no locks are held on any di after the site recovers.

As far as updating the value of a data value di that is affected by committed transactions is concerned, the logs need to be accessed to determine which transactions committed but did not update the database prior to the site failure. Those updates are re-done. The re-doing actions must be idempotent [4] in view of the

365
possibility of a failure during the recovery phase. Note that if desired, by using check-pointing mechanisms, the number of re-do actions required may be reduced in the usual manner \([4]\).

Outstanding \(Vm\) need not be sent again — this is handled by the message protocol. The system eventually sends the outstanding \(Vm\) in the normal course of processing. With regard to link failures, as discussed in Section 4, no special actions need be taken. The protocol for message handling that we use, and the time out counters being used in the transaction processing together ensure that the system is not adversely affected by such behavior.

**Theorem 6:** The DvP and \(Vm\) scheme provides an independent recovery mechanism for an arbitrary number of site failures. \(\square\)

### 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 \(Vm\).

Our newly proposed scheme alleviates the problems of correctly managing data in a failure-prone environment. Our scheme has the advantage of simplicity, non-blocking transaction processing, a uniform approach to the handling of different kinds of failures, and independent recovery from site failures. Furthermore, in the case of network partitions, there is still the possibility of continuing the normal operations, thereby allowing higher accessibility. The correctness criterion that we have outlined is based on the acceptable notion of serializability.

DvP and \(Vm\) can be applied profitably in applications such as inventory control, airline reservations, and banking applications. 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. DvP can also find use in applications where it is not possible to carry-out processing to ascertain the causes of failure, or where it is imperative to minimize the contention for certain types of resources. For example, it may be interesting to consider the use of DvP in systems that use “aggregate fields” ([7] and references cited therein) since the aggregate fields have partitionable values.

Our scheme cannot, in its current form, be applied to all database applications. However, as we remarked, it can be effectively used in a variety of applications. Notice that the description of transaction execution indicates that the access set \(A(t)\) of a transaction \(t\) must be known at the start of its execution. This involves some performance penalty since some data items that a transaction does not actually access (e.g., due to a conditional statement) are, nevertheless, locked by the protocol. Also, there is a high overhead in accessing the entire value of a particular data item, but it should be noted that in case of failures, traditional environments may not permit reading at all if updating of data items is permitted at several locations. In our scheme, the failure of a site implies the inaccessibility of the information exclusively residing there; but the same problem exists in a traditional non-replicated database. In a replicated database, it would not be possible to distinguish between site failures and network partitions. In such situations, it would be unsafe to access the information, except to read, at any location. 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, while the rest still remains accessible.

One question that arises in any system that has requests and waiting periods is how to handle deadlock and live-lock situations. Deadlock is characterized by wait-for graphs that contain cycles which are stable ([8, 4]). Since all waiting periods in our scheme are time-bound, there is no situation in which any wait-for graph is stable (with or without cycles). Thus, in the strict sense, the question of deadlocks does not arise. However, our scheme, as described, may suffer live-locks. Any system that employs locking and waiting periods needs to deal with live-locks. As a result, it is a well-studied problem, and several solutions exist [8, 4] that could be incorporated into our scheme.

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 these 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. The answer may lie in radically different approaches to the solution of the problems. This paper should be viewed as a step in this direction.

It should be noted that several optimizations may be possible to improve the efficiency of the system. For example, different locking modes could be introduced, or more optimism could be allowed in the protocol for concurrency. The properties of partitionable operators and representations could yield several other optimizations.
Also, if the system has more favorable failure characteristics, it may be possible to find other improvements. For more general systems, some hybrid of DvP and traditional approaches could also be considered. Another improvement that may be explored is the use of nested transaction techniques [9] 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 RD transactions is similar to an application of nested transaction methods. Of course, nested transactions increase the potential for the parallel execution of transactions even locally at each site.

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, to our knowledge, the notions of consistency and serializability that we use are 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.

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 is worth studying the ideas further especially because the handling failures in a distributed system is known to be a difficult problem.

Two important aspects of the scheme need further examination. Firstly, there is a need to find ways to extend the methods to handle more data types. Secondly, performance studies are needed to find the best ways to distribute the data, to design the transactions and to reduce the message traffic.

References


