A Formal Model of Crash Recovery in a Distributed System

DALE SKEEN AND MICHAEL STONEBRAKER

Abstract—A formal model for atomic commit protocols for a distributed database system is introduced. The model is used to prove existence results about resilient protocols for site failures that do not partition the network and then for partitioned networks. For site failures, a pessimistic recovery technique, called *independent recovery*, is introduced and the class of failures for which resilient protocols exist is identified. For partitioned networks, two cases are studied: the pessimistic case in which messages are lost, and the optimistic case in which no messages are lost. In all cases, fundamental limitations on the resiliency of protocols are derived.

Index Terms—Commit protocols, crash recovery, distributed database systems, distributed systems, fault tolerance, transaction management.

I. Introduction

IN THIS PAPER we present a formal model for transaction processing in a distributed database and then extend it to model several classes of failures and crash recovery techniques. These models are used to study whether or not resilient protocols exist for various failure classes.

Crash recovery in distributed systems has been studied extensively in the literature [2], [4]-[6], [9], [11], [14]-[16]. Many protocols have been designed that are resilient in some environments. All have an "ad hoc" flavor to them in the sense that the class of failures they will survive is not clearly delineated.

The purpose of this paper is to formalize the crash recovery problem in a distributed database environment and then give some preliminary results concerning the existence of resilient protocols in various well-defined situations.

Consequently, in the next section we give a brief introduction to transactions in a distributed database. In Section III we indicate the assumed network environment and our model for transaction processing. In Section IV we extend the model to include the possibility of site failure and give results concerning the existence of resilient protocols in this situation. Section V turns to the possibility of network failure and shows the class of failures for which a resilient protocol exists. Section VI summarizes the results in the previous two sections.

Manuscript received November 10, 1980; revised December 14, 1982. This work was supported by the U.S. Air Force Office of Scientific Research under Grant 78-3596, the U.S. Army Research Office under Grant DAAG29-76-G-0245, and the Naval Electronics Systems Command under Contract N00039-78-G-0013.

D. Skeen is with the Department of Computer Science, Cornell University, Ithaca, NY 14850.

M. Stonebraker is with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

II. Background

A distributed database management system supports a database physically distributed over multiple sites interconnected by a communications network. By definition, a *transaction* in a distributed database system is a (logically) atomic operation: it must be processed at all sites or at none of them. Designing protocols resilient to various failures, including arbitrary site failures and partitioning of the communications network, is a very difficult task.

Preserving transaction atomicity in the single-site case is a well-understood problem [4], [7]. The processing of a single transaction is viewed as follows. At some time during its execution, a "commit point" is reached where the site decides to commit or to abort the transaction. A *commit* is an unconditional guarantee to execute the transaction to completion, even in the event of multiple failures. Similarly, an *abort* is an unconditional guarantee to back out the transaction so that none of its effects persist. If a failure occurs before the commit point is reached, then immediately upon recovery the site will abort the transaction. Both commit and abort are *irreversible*.

In the multiple site case, it is the task of a *commit protocol* to enforce global atomicity. Assuming that each site has a recovery strategy that provides atomicity at the local level, the problem becomes one of ensuring that the sites either unanimously abort or unanimously commit the transaction. A mixed decision results in an inconsistent database. In the absence of failures, a unanimous consensus is easily obtained by a simple protocol. The challenge then is to find protocols ensuring atomicity in the presence of inopportune and perhaps repetitive failures.

A basic assumption within this paper is that during the initial phase of distributed transaction processing any participating site can unilaterally abort the transaction. A site may choose to abort for any of the following reasons:

1) one or more sites fail,
2) the network fails,
3) the transaction deadlocks with another transaction,
4) the user aborts the transaction.

Clearly, before any site can commit the transaction, all sites must relinquish their right to unilaterally abort it. Once a site has relinquished that right, it can abort the transaction only in concordance with the other sites.

Let us now examine a commit protocol allowing sites to unilaterally abort. One of the simplest and certainly the most renowned is the *two-phase commit protocol* [4], [6] illus-

0098-5589/83/0500-0219$01.00

Site 1 (coordinator)	Site 2 (slave)
(1) Transaction is received. "Start Xact" is sent.	
	"Start Xact" is received. Site 2 responds: "yes" to commit, "no" to abort. The response is sent to Site 1.
(2) Site 2's response is received. If response = "yes" and Site 1 agrees, then "commit" is sent; else, "abort" is sent.	
	Either "commit" or "abort" is received and processed.

Fig. 1. The two-phase commit protocol (two sites).

trated in Fig. 1 for two sites.[1] It is a centralized protocol with a single coordinator (Site 1) and with the remaining sites acting as slaves.

In the first phase, the coordinator receives the transaction and forwards it to the slave (Site 2). The slave then partially executes the transaction, and indicates its readiness to commit ("yes") or its intent to unilaterally abort ("no"). The commit decision is made by the coordinator after receiving a response from the slave, and a "commit" message is sent only if both agree to process the transaction. For a transaction to commit, three messages are exchanged: "start transaction" is sent to Site 2; "yes" is sent to Site 1; and finally "commit" is sent to Site 2. Although the protocol is illustrated for only one slave, any number of slaves may participate.

Clearly, the protocol is correct in the absence of failures. Its fundamental weakness is its vulnerability to site failures, especially failure of the coordinator. When the coordinator fails, the remaining sites must block transaction execution until it recovers.

It is always an option for a distributed database system to block whenever a failure occurs. Even though blocking preserves consistency, it is highly undesirable because the locks acquired by the blocked transaction cannot be relinquished. rendering the data inaccessible by other requests. Consequently, the availability of data stored at reliable sites can be limited by the availability of the weakest component in the distributed system. For this reason we postulate that blocking protocols are unacceptable to many applications.

In this paper we confine our interests exclusively to *nonblocking protocols*—protocols in which operational sites never suspend because of a failure. We say that a commit protocol is *resilient* to a class of failures only if the protocol enforces transaction atomicity and is nonblocking for any failure within that class. The nonblocking constraint guarantees that a resilient protocol will always terminate irrespective of the frequency of failures. A necessary but not sufficient condition for nonblocking behavior is a strict bound on the number of messages sent by a resilient protocol.

[1]The two-phase commit protocol depicted is an optimized version of the standard two-phase commit protocol. The standard protocol includes "ack's" following the receipt of as commit or abort by the slaves. These messages are convenient for bookkeeping purposes but add nothing to the fault tolerance of the protocol and thus have been deleted.

III. The Transaction Model

In this section we introduce the model, ignoring the effects of failures. Sites failures and network failures are introduced into the model in Sections IV and V, respectively.

A. The Network Assumptions

The network provides point-to-point communication between any pair of sites. It is assumed to have the following characteristics:

1) it delivers a message within a preassigned time period T, or

2) it reports a "timeout" to the sender.

When a timeout occurs, the sender can safely assume that the network or the recipient or both has failed. In the case of a network failure, it is not known whether the recipient received the message.

Notice that we are assuming a somewhat idealized environment by precluding the possibility that a timeout is caused by a slow but correctly executing site. However, by adjusting T and by providing low-level protocols for verifying failures, we can build systems that differentiate between failures and slow responses with an arbitrarily high degree of confidence.

B. Specifying a Protocol

Reasoning about commit protocols requires a well-defined notion of the "state" of the transaction at each participating site. Broadly speaking, this abstract state is a concise summary of transaction history and, hence an indicator of the options available to a recovery protocol. In the above description of transaction processing, we informally discussed three such states: the initial state where sites have the right to unilaterally abort, the abort state, and the commit state. We now formalize the notion of state by borrowing from classical automata theory.

The formal specification of a protocol consists of a collection of nondeterministic finite state automata (FSA)—one for each site. The automaton executing at Site i is called the *local protocol* for Site i. The state of this automaton is the *local transaction state* (or, more succinctly, the local state) for Site i. The network is modeled as a completely passive device. It is an unbounded buffer that serves as a common read/write medium for all local protocols. A local state transition consists of reading one or more messages, perfoming some local pro-

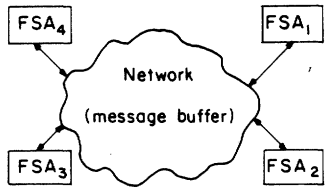

Fig. 2. The model with four sites.

cessing, and sending zero or more messages. Fig. 2 illustrates four sites participating in a distributed transaction. Similar models have been used for network communication protocols [1], [3].

There are several restrictions on this collection of FSA's.

1) The FSA's are nondeterministic. The behavior of each FSA is not known *a priori* because of the possibility of deadlocks, failures, and user aborts. Moreover, when multiple messages are addressed to a site, the order of receiving the messages is arbitrary.

2) The final states of the FSA's are partitioned into two sets: the "abort" states A and the "commit" states C.

3) There are no transitions from a state in A to a state not in A. Similarly, there are no transitions from a state in C to a state not in C. Therefore, once a site enters an "abort" state ("commit" state), the site remains in such a state. This corresponds to the requirement that abort and commit are irreversible operations.

4) The state diagram defining an FSA is acyclic. This suffices to guarantee that a protocol sends a bounded number of messages.

State transitions are assumed to be asynchronous among the sites and, in the absence of failures, atomic. It is convenient to consider a transition to be an instantaneous event.

The local protocols for the two-site, two-phase commit protocol (Fig. 1) are illustrated in Fig. 3.

The state diagram illustrates the conventions used in the remainder of the paper. The states for Site i are subscripted by i, and final states are doubly circled. Messages received during a state transition are shown above the horizontal line; messages sent are shown below the line.

Both local protocols begin processing in their initial states (q_1 and q_2). A transaction begins when a *request* message is received from the application program. The receipt of the request by the coordinator (Site 1) causes a state transition to state w_1 (the wait state) and the sending of the transaction to Site 2. Upon receipt of the transaction, Site 2 nondeterministicly choses one of two possible replies. Either it replies with a "yes," accepting the transaction and moving into p_2 (the prepared or precommit state), or it replies with a "no," unilaterally aborting the transaction and moving directly to a_2. The protocol continues until both sites occupy final states: either commit (c) or abort (a).

C. The Global Transaction State

The *global state* of a distributed transaction is defined to consist of:

1) a global state vector containing the states of the local protocols,

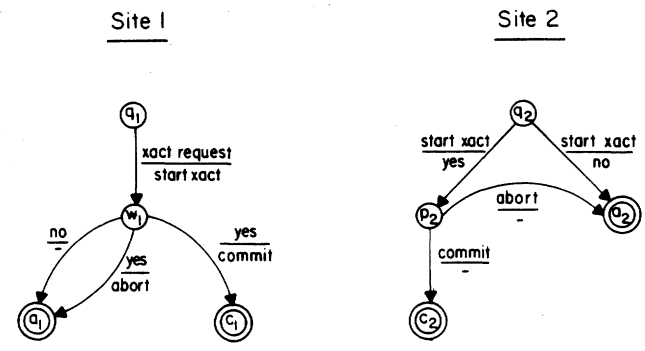

Fig. 3. The local protocols for the two-phase commit.

2) the outstanding messages in the network.

The global state defines the complete processing state of a transaction.

A global state is a *final state* if all local states contained in its state vector are final states. It is said to be *inconsistent* if its state vector contains both a commit state and an abort state.

A *global state transition* occurs whenever a local state transition occurs at a participating site. Barring failures, this is the only time that global state transitions occur. Since local state transitions are viewed as instantaneous and hence mutually exclusive events, exactly one local transition occurs during a global transition.

If there exists a global transition from global state G to global state G' then G' is said to be *immediately reachable* from G. A global state, together with the definition of the protocol, contains the minimal information necessary to compute all of its immediately reachable states. Starting with a unique initial global state and taking the transitive closure of the immediately reachable states, we obtain all of the reachable states. The reachable global state graph for the (2-site) two-phase commit protocol are illustrated in Fig. 4. In the graph the descendents of a state are its immediately reachable successors.

The global state graph is an invaluable tool in both analysis and verification of the protocol, graphically rendering all possible actions of the protocol. A path from the initial state to a terminal state (i.e., a state without a successor) corresponds to a possible execution sequence of the protocol. The graph itself is easy to generate automatically, but can be quite large (exponential in N, the number of sites). Fortunately, a small N usually serves to illustrate a protocol, and proofs seldom require the generation of the entire graph.

A protocol is *operationally correct* only if its reachable state graph contains no inconsistent states and all terminal states are final states. When a graph contains terminal states that are not final states, then it is possible for some sites to never commit or abort the transaction. Applying these definitions to the state graph of Fig. 4, we can quickly verify the correctness of the two-phase protocol.

C. The Concurrency and Sender Sets

Two local states are said to be *potentially concurrent* if there exists a reachable global state containing both local states. Thus, for at least one possible execution of the protocol, the

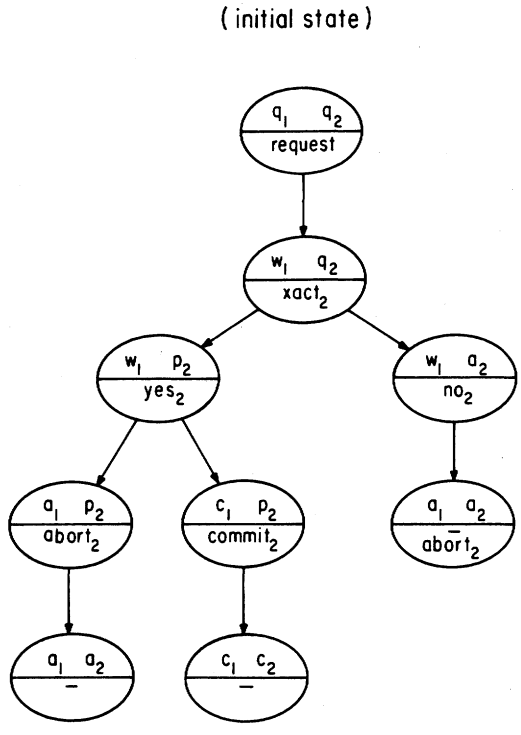

Fig. 4. Reachable state graph for the two-phase commit.

first state is occupied by one site at the same time that the second state is occupied by another site.

We now define two sets that will be used extensively in subsequent proofs. Both sets are easily constructed from the global state graph.

Definition: Let s be an arbitrary local state. The *concurrency set* of a local state s is the set of all local states that are potentially concurrent with it. We denote this set by $C(s)$.

From the reachable state graph given in Fig. 4, we observe that the concurrency set for w_1 consists of $\{q_2, a_2, p_2\}$.

When a site makes a transition from state s to state t, it is convenient to consider the messages received and sent during the transition as being "received" and "sent" by state s. For example, in the two-phase commit we would say that the coordinator's wait state w_1 sends commit messages that are received by the slave's wait state p_2. We will be interested in the set of local states sending messages to a given state s.

Definition: Let s be an arbitrary local state, and let M be the set of messages that are received by s. The *sender set for s*, denoted $S(s)$, is $\{t \mid t$ "sends" m and $m \in M\}$.

Again referring to Fig. 4, we observe that $S(w_1)$ is $\{q_2\}$.

IV. Site Failures

This section examines existence questions on nonblocking protocols for site failures. Here, we assume that the network never fails—an assumption that is relaxed in the next section.

A. Modeling Site Failures

A failure of any type is normally detected by the absence of an expected message. We assume that each site has at its disposal an interval timer allowing it to bound the time it waits for the receipt of a message. When the timer expires, the site is said to have "timed out" and may take appropriate action. This is modeled by *timeout* messages that are received like any other message and can cause a state transition.

Site failures are modeled by a *failure transition*, which is a special kind of local state transition [10]. Such a transition occurs *at the failed site* the instant that it fails. The resulting local state is the state initially occupied by the failed site upon recovering. An underlying assumption is that a site can detect when it has failed.

Let us temporarily assume atomic state transitions even in the presence of failures. Hence, a failure cannot occur during the middle of a transition and interrupt the sending of messages. In this case a failure transition can be simply defined—it reads all outstanding messages and sends a *timeout* message to all participants.

In real systems, state transitions are not atomic and sites can fail after sending only a few of the messages associated with a transition. This can be modeled simply by allowing a failure transition to send any prefix of the messages normally sent by a valid transition from the same state. These messages are sent in addition to the timeout messages sent by the failure transition. We shall also allow failure transitions from the same state to terminate in different states. This generalization of failure transitions is sufficiently powerful to accurately model the behavior of any implementation of an FSA.

B. Independent Recovery

Independent recovery refers to a scheme where a recovering site makes a transition directly to a final state without communicating with other sites. Only local state information is used in the recovery protocol, hence recovery is independent of any event occurring after the site's failure. Independent recovery is modeled by requiring that all failure transitions terminate in a final state. This final state is assumed by the site immediately upon recovering.

Independent recovery is interesting for several reasons. First, it is easy to implement and leads to simple protocols. One need not be concerned with messages to a failed site being queued in the network or at another site that may be down when the failed site attempts to recover. This recovery strategy is of theoretic interest because it represents the most pessimistic recovery strategy. Proving the existence of a class of resilient protocols using independent recovery implies the existence of resilient protocols in all more sophisticated strategies of site failures. Its most important practical aspect is that it qualifies the usefulness of local state information during recovery. If the local state proves to be insufficient for resilient recovery then those sites remaining operational during the duration of the transaction must maintain the transaction's outcome for the inoperative sites. This history will have to be maintained indefinitely until all sites have recovered and completed the transaction. In this regards, independent recovery provides the only true nonblocking recovery strategy—any strategy requiring a history mechanism will necessarily block when the history becomes temporarily unavailable due to failures.

When discussing independent recovery, we will restrict our attention to the two site case. All of the results are easily extended to the multisite case. The general case is not pre-

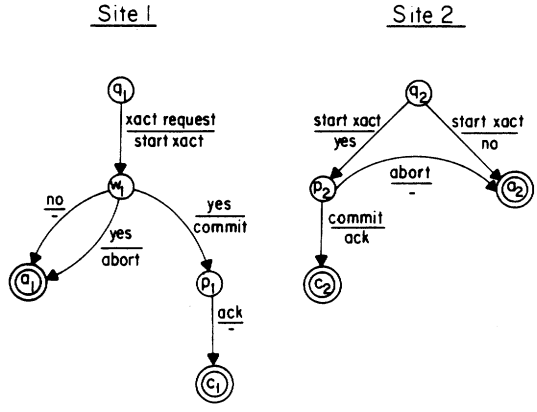

Fig. 5. The two-phase commit protocol extended with an *ack* message.

sented here because additional notation is required and the resulting protocols are of little practical importance.

C. Failure of a Single Site

Let us first consider the simple case where at most one site fails during a transaction. Our goal is to develop rules for assigning failure and timeout transitions to existing protocols to form protocols resilient to single site failures.

Not all protocols can be made resilient, as the next lemma demonstrates.

Lemma 1: If a protocol contains a local state with both abort and commit in its concurrency set, then under independent recovery it is not resilient to an arbitrary single failure.

Proof: This follows directly from the definition of "concurrency set." Consider a local state s_i, and its concurrency set $C(s_i)$. Let $C(s_i)$ contain both an abort state and a commit state. Clearly, s_i cannot have a failure transition to the commit state, since the other site may be in the abort state. Similarly, s_i can not have a failure transition to the abort state, since the other site may be in the commit state. Hence, when Site i is in s_i, it cannot safely and independently recover. □

If a protocol has no local states violating the necessary condition in the above lemma, then failure transitions can be assigned according to the following rule.

Rule 1: For every intermediate state s in the protocol: if $C(s)$, contains a commit, then assign a *failure* transition from s to a commit state; otherwise, assign a *failure* transition from s to an abort state.

The two-phase commit protocol does not satisfy the condition in the lemma: the concurrency set of the slave's prepared state (p_2) contain both c_1 and a_1. However, p_2 is the only local state violating this rule. This occurs because the coordinator moves into the commit state before the slave acknowledges committing the transaction. If instead the coordinator moves to a *prepared* state while it is waiting for the acknowledgment from the salve and moves into a commit state only after the acknowledgment is received, then it is possible to assign a failure transition to p_2. This "extended" two-phase commit protocol is shown in Fig. 5 and its (reachable) global state graph is shown in Fig. 6.

From the graph, it is easy to verify that the concurrency set for each state, including the *prepared* state, contains only one kind of final state. Hence, failure transitions satisfying Rule 1

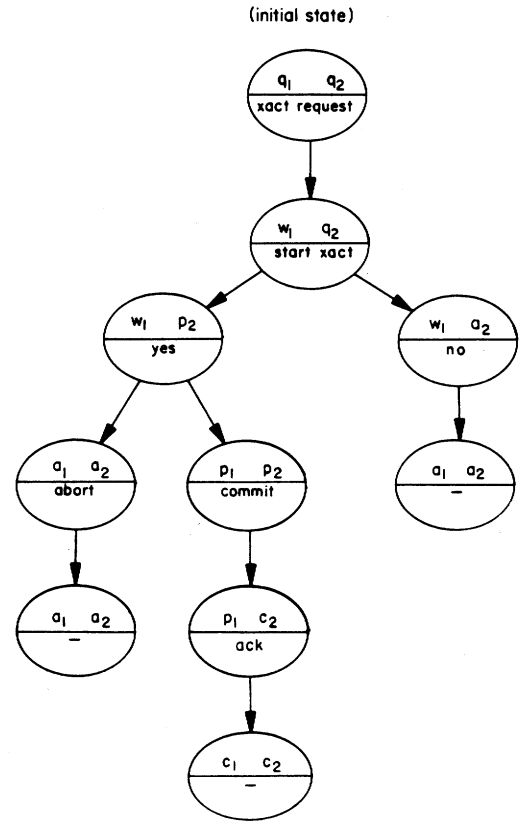

Fig. 6. The reachable global state graph for the protocol of Fig. 5.

can be defined for each state. The assignment of failure transitions is depicted in Fig. 7 (timeout transitions, to be discussed subsequently, are also illustrated).

The second rule deals with timeout transitions.

Rule 2: For each intermediate state s_i: if t_j is in $S(s_i)$ (the *sender set* for s_i) and t_j has a failure transition to a commit (abort) state, then assign a timeout transition from s_i to a commit (abort) state.

This rule is less obvious than the previous one. A "timeout" can be viewed as a special message sent by a failed site in lieu of a normal message. Like any other message received by state s_i, it must have been "sent" by a state in the sender set for s_i. Moreover, the failed site, using independent recovery, has made a failure transition to a final state. Hence, the receiving state must make a consistent decision. Timeout transitions for the extended commit protocol are illustrated in Fig. 7.

The protocol displayed is resilient to a single failure by either site. This can be verified by examining its reachable state graph. In fact, the rules always yield a resilient protocol under independent recovery.

Theorem 2: Rules 1 and 2 are sufficient for designing protocols resilient to a single site failure.

Proof: Let P be a protocol with no local states having a concurrency set containing both a commit state and an abort state. Let P' be the protocol resulting from assigning failure and timeout transitions to P according to the above rules. The proof proceeds by contradiction.

We will assume that P' is not resilient to all single site failures. Therefore, there must exist a path from the initial global state to an inconsistent final global state, and this path con-

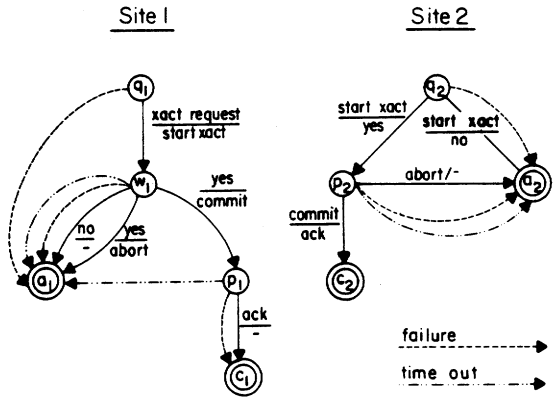

Fig. 7. The protocol with failure and timeout transitions obeying Rules 1 and 2.

tains exactly one failure. Without loss of generality, assume Site 1 fails, and assume that it fails in state s_1. Let the inconsistent global state contain the final states f_1 and f_2. Hence, upon failing, Site 1 made a failure transition from s_1 to f_1. There are two cases depending on whether Site 2 is in a final state or a nonfinal state when Site 1 fails.

Case 1: Site 2 is in the final state f_2. But this implies that f_2 is in $C(s_1)$. Therefore, rule 1 is violated.

Case 2: Site 2 is in a nonfinal state. Upon failing, Site 1 sends a "timeout" that is received by Site 2 while in state s_2. Now, Site 2 makes a timeout transition to f_2 that is inconsistent with f_1. However, by definition, s_1 is in the sender set of s_2. Therefore, Rule 2 has been violated. □

D. Two Site Failures

The rules given above are sufficient for protocols resilient to a single failure; however, such protocols are not necessarily resilient to the failure of two sites. This can be demonstrated for the protocol of Fig. 3. Double failures occurring when Site 1 is in state p_1 and Site 2 is in state p_2 results in an inconsistent final state. Unfortunately, this is the only possible assignment of failure and timeout transitions satisfying both rules; hence, this protocol cannot be made resilient to two failures using only Rules 1 and 2.

Recall that the addition of a single state to the two-phase commit resulted in the above protocol, which enjoys a marked increase in resiliency over its progenitor. Can this protocol be extended with additional states to deal with double failures? The next theorem yields a negative answer.

Theorem 3: There exists no protocol using *independent recovery* that is resilient to arbitrary failures by two sites.

The double failures that are impossible to handle are "concurrent failures"—failures occurring close enough together such that neither site detects the failure of the other before itself failing. Although we will not prove it, concurrent failures are the only class of double failures from which it is impossible to recover. We now briefly sketch the nonexistence argument assuming double concurrent failures. In the discussion, we use the shorter term "j independently recovers to state s from G" instead of the more precise term "Site j independently recovers to state s after failing while the global state is G."

Let P be a commit protocol, and consider an execution of P resulting in global commit. This execution corresponds to some path, $G_0, G_1, \cdots, G_m$, in the global state graph for P. Clearly, from the initial global state G_0, all sites independently recover to the abort state, and from the global commit state G_m, all sites independently recover to the commit state. Let G_k be the first global state where a site recovers to a commit state, and let j be such a site. Since j independently recovers to the abort state in G_{k-1} (by assumption) but not in G_k, j must have changed states in the global state transition from G_{k-1} to G_k. Moreover, j was the only site to make a state transition (see Section III-C). Hence, the other sites occupy the same states in G_k as they did in G_{k-1}, and each must therefore independently recover to the same state in both G_k and in G_{k-1}, namely, the abort state. Consequently, the failure of j coupled with any other failure while the global state is G_k results in an inconsistent state.

The above argument applies not only to the two-site case but to any number of sites. The complete proof for $N (N > 1)$ sites appears in the Appendix.

Several conclusions can be drawn from these results. Since a site cannot determine concurrent failures from its local state, it cannot determine when independent recovery can be safely used. For a two site protocol this means that in all cases a recovering site must block until it can communicate with its cohort. Moreover, the cohort must maintain a record of the transaction's outcome if the transaction completed.

Although it is not possible to design a protocol that allows nonblocking recovery, it is possible to design commit protocols that never require an operational site to block. (Here "operational site" refers to a site that has not failed since the beginning of the transaction.) Such protocols are of great practical importance: examples include the four-phase protocol of SDD-1 [5] and the family of three-phase protocols [12]. All of these protocols require that a recovering site poll the other sites about the status of any outstanding transaction.

V. Network Partitioning

A network failure results in at least two sites that cannot communicate with each other. We model such a partition in two ways. The first model is a pessimistic model where all messages are lost at the time partitioning occurs. The second model is an (overly) optimistic model where no messages are lost at the time partitioning occurs; instead, undeliverable messages are returned to the sender. While the pessimistic model is more realistic, the optimistic model is theoretically interesting since it yields upper bounds on the achievable resiliency.

A *simple partitioning* occurs when the sites are partitioned into exactly two sets with no communication possible between the sets. A *multiple partitioning* occurs when the sites are partitioned into $k (k > 2)$ sets. A multiple partitioning can be viewed as simultaneous occurrences of two or more simple partitionings.

A protocol is *resilient* to a network partitioning only if it is *nonblocking*, that is, the protocol must ensure that each isolated group of sites can reach a commit decision consistent with the remaining groups. Since the commit decision within

a group is reached in the absence of communication to outside sites, this problem is very similar to the independent recovery paradigm presented in the previous section.

Unless otherwise stated, we assume that partitions are caused by link failures rather than site failures.

A. Partitioning With Loss of Messages

In this pessimistic case, a network partitioning is modeled as a special type of global state transition. Until now all global state transitions have been triggered by one local state transition. However, a network partitioning is modeled as a global state transition that changes the network state while leaving all local states unchanged. Specifically, a partitioning erases all outstanding messages and substitutes timeouts in their place. As before, upon reading a timeout message, a site may make a "timeout" transition.

Let us first examine the simplest type of partitioning: a simple partitioning where each partition contains exactly one site. This situation is analogous to a double site failure in a two site protocol using independent recovery. The difference is that when a double failure occurs, sites make "failure" transitions; whereas, when a partitioning occurs, sites make "timeout" transitions. It can be shown that a solution to the double failure problem implies a solution to the simple partitioning problem. An immediate consequence of this observation is the next theorem.

Theorem 4: There exists no protocol resilient to a network partitioning when messages are lost.

This applies to multiple partitionings as well as simple partitionings. The Appendix contains a proof outline that uses the proof of Theorem 3 as a paradigm.

Any time that a partitioning results in lost messages or there is the possibility of lost messages, a blocking protocol must be used. Unfortunately, this is the normal situation, especially for wide area networks. In the worst case only a single partition can continue processing.

Finally, let us consider the possibility of site failures in conjunction with network partitioning. If all participants can differentiate between site failures and partitioning, then there is no problem. On the other hand, if a group of communicating sites cannot determine conclusively the type of failure that has occurred, then they must assume the worst, which is a partitioning. In the special case when there are only two sites, it is senseless to run any protocol more elaborate than the two-phase commit unless both sites can frequently differentiate between a site failure and a node failure. When they cannot differentiate, one site (presumably the slave) will be forced to block anyway.

B. Partitioning with Return of Messages

In this environment we assume that the network can detect the presence of a partition and return undeliverable messages to their senders. This appears to represent the most optimistic model for partitions, while loss of messages is the most pessimistic one.

In this optimistic case a partition causes a global state transition that redirects all undeliverable messages back to their senders and writes timeout messages to the recipients of undeliverable messages. A site may make a transition whenever an undeliverable message is returned to it or a timeout is received.

C. The Optimistic Two-Site Case

To study this optimistic situation, we now define two design rules that resilient protocols must satisfy.

Rule 3: For a state s_i: if its concurrency set, $C(s_i)$, contains a commit (abort) state, then assign a *timeout* transition from s_i to a commit (abort) state.

Here, Site i in state s_i was expecting a message when the partition occurred. Instead, it received a "timeout." This site will then make a decision to abort or commit the transaction consistent with the state of the sender of the undeliverable message. At this time the site can infer that any message it sent arrived at its destination but any reply to that message was returned undelivered.

The second rule deals with a site sending an undeliverable message. It must make a commit decision consistent with the decision of the intended receiver.

Rule 4: For state s_j: if t_i is in $S(s_j)$, the sender set for s_j, and t_i has a *timeout* transition to a commit (abort) state, then assign a *undeliverable message* transition from s_i to a commit (abort) state upon the receipt of an undeliverable message.

An observant reader will note that these rules are equivalent to the rules given for independent recovery of failed sites. In fact, the two models are isomorphic. To illustrate the equivalence, consider the information conveyed by a "timeout" message from a failed site. The following is true when the operational site i receives the "timeout" indicating a failure of the other site:

1) the last message sent by Site i was not received (the other site failed prior to its receipt),

2) communication with the other site is impossible (it is down),

3) the other site will decide to commit using independent recovery.

Exactly the same conditions hold when an undeliverable message is returned to Site i.

Applying the above design rules to the protocol of Fig. 5 yields the protocol illustrated in Fig. 8. As expected, the protocol is isomorphic to the protocol of Fig. 7.

In light of this isomorphism, Theorem 5 is not surprising.

Theorem 5: Design Rules 3 and 4 are necessary and sufficient for making protocols resilient to a partition in a two-site protocol.

Sketch of Proof: We can make use of the proof of Theorem 2. In that proof, substitute "undelivered message" for every occurrence of "timeout" and substitute "timeout" for every occurrence of "failure." Finally, substitute "Rule 3" for "Rule 1" and "Rule 4" for "Rule 2." The result is a proof for Theorem 5. □

An implicit assumption made above is that a site can distinguish between a timeout resulting from a site failure and a timeout resulting from a partitioning. Comparing the protocol in Fig. 8 to the protocol in Fig. 6, we note that the coordinator's behavior in state p_1 is dependent on the cause of the timeout. Hence, for this protocol, knowing the cause of the

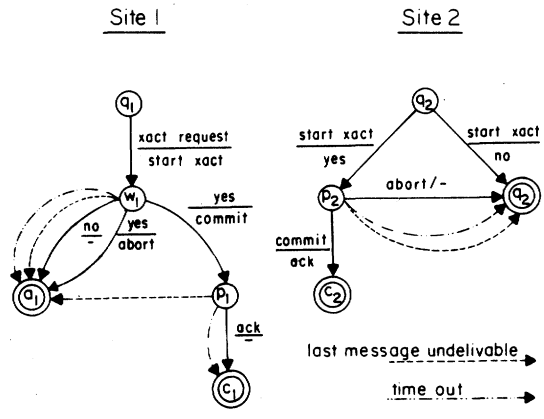

Fig. 8. The extended two-phase commit protocol (of Fig. 5) augmented with *timeout* transitions and "undeliverable message" transitions according to Rules 3 and 4.

failure (at least for the coordinator) is crucial. It is not hard to show that this is always the case. Of course, within this model, detecting the cause of a timeout is easy to finesse: if a site is uncertain as to the reason for a timeout, then it can send another message to the same site. Presumably, if the network is down, then it will return the message "undelivered" rather than a "timeout."

D. The Optimistic Multisite Case

In the absence of site failures, the multisite case is very similar to the two-site case, since preserving consistency within a connected group of operational sites is not difficult. Thus, design rules 3 and 4 can be extended to multisite protocols in a straightforward way. This leads to the following result.

Corollary 6: There exist multisite protocols that are resilient to a *simple* partition when undeliverable messages are returned to the sender.

This result is the complement of the results obtained from the pessimistic model discussed earlier. The models differ in their handling of outstanding messages when the network fails: in the pessimistic model, they are lost; whereas, in the optimistic model, they are returned to their sender. Since this is the only difference between the two models, the next result is implied.

Corollary 7: Knowledge of which messages were undelivered at the time the network fails is necessary and sufficient for recovering from simple partitions.

We now turn to multiple partitions. Since we are dealing with an optimistic situation, we assume that timeouts and undeliverable messages are unaffected by additional partitions. This, in effect, is an assumption that the network is partitioned into all subsets simultaneously.

Even in this (overly) optimistic model, our results are negative, which implies negative results for all realistic partitioning models.

Theorem 8: There exists no protocol resilient to a multiple partition.

Therefore, even complete information about message traffic during a partition, and in particular, information about which messages are undeliverable, is insufficient for recovering from multiple partitions.

The proof of this theorem is similar to the proof of Theorem 3, but it is somewhat more complicated since the network state in addition to the local transaction states must be examined. The proof appears in the Appendix.

VI. Conclusions

The major contribution of this paper is the formal model for atomic commit protocols based on nondeterministic finite state machine. Nondeterminism is used in modeling unpredictability in the environment, including the order of message deliveries and failures. Herein, we have used the model to study the existence of nonblocking protocols for site and network failures. Companion papers have used it for specifying and verifying several families of very resilient protocols [11], [12]. It has been our experience that the model provides a convenient conceptual framework for reasoning about protocols.

The results in Sections IV and V define fundamental limitations on the robustness of protocols with respect to both site failures and network partitions. Many of the limitations have been part of the accepted folklore on fault tolerant distributed computing. For this reason, the results tend to be more illuminating than surprising. However, we believe that this is the first formal, systematic treatment of such existence questions. The approach is extensible to problems in related areas, such as existence results for atomic broadcast.

The only truly nonblocking site recovery strategy, independent recovery, uses local state information available at failure time. Hence, recovery does not depend on messages to down sites being queued in the network or on operational sites maintaining a log of completed transactions. This strategy is resilient to a single failure but no more. From the nonexistence proof for the case of two failures, it is clear that concurrent failures are the most difficult failures to handle: in general, the database is left in an inconsistent state if independent recovery is attempted after concurrent failures. Since a site cannot deduce from its local state whether another site's failure was concurrent with its failure, *a recovering site cannot determine when it is safe to use independent recovery*.

The results on robust network protocols are more discouraging than the independent recovery results. In realistic network environments, where a partition can result in lost messages, there exists no nonblocking protocols—not even for "simple" partitions. Therefore, in the worst case, the best protocols allow no more one group of sites to continue while the remaining groups block.

Appendix

Herein, the complete proofs for Theorems 3, 4, and 8 are given.

Theorem 3: There exists no protocol using *independent recovery* of failed sites that is resilient to two-site failures.

Proof: Let P be a protocol that always preserves consistency in the absence of failures. Let i and j be two sites. We will show: for every failure-free execution of P that commits the transaction, there exists a point in the execution where a failure of i followed by a failure of j leads to an inconsistency.

A failure-free execution of P corresponds to a path in the global state graph, $G_0, G_1, \cdots, G_m$ where G_0 is the initial

global state and G_m is a final global state. We are assuming that G_m is a final *commit* state. A global state consists of a local state vector and a network state; however, for the remainder of this proof, we will ignore the network state. Hence, we view the global state G_k as a vector $<s_{k1}, s_{k2}, \cdots, s_{kN}>$, where s_{ki} is the local state for Site i when transaction execution is in state G_k. N is the number of participating sites.

Let $f(s)$ be the result of making a failure transition while in state s. Let F_k be the global state resulting from the double failure of i and j when transaction processing is in G_k. F_k is equal to G_k except that the s_{ki} and s_{kj} are replaced by $f(s_{ki})$ and $f(s_{kj})$.

Let us now examine the sequence $F_0, \cdots, F_m$ and, in particular, the local states for i and j in this sequence. The pair $(f(s_{0i}), f(s_{0j}))$ must be equal to (a_i, a_j) since a site will always abort the transaction when it fails in the initial state. Similarly, the pair $(f(s_{mi}), f(s_{mj}))$ must equal (c_i, c_j) since we have assumed that the transaction was committed. Let k be the smallest k such that either $f(s_{ki})$ or $f(s_{kj})$ yields a commit state. This situation is depicted below:

Commit Sequence (G_I)	Global State (F_I) Resulting from the Failure of i and j
$<\cdots, s_{0i}, \cdots, s_{0j}, \cdots>$	$<\cdots, a_i, \cdots, a_j, \cdots>$
$\vdots$	$\vdots$
$<\cdots, s_{k-1,i}, \cdots, s_{k-1,j}, \cdots>$	$<\cdots, a_i, \cdots, a_j, \cdots>$
$<\cdots, s_{ki}, \cdots, s_{kj}, \cdots>$	F_k where either $f(s_{ki})$ or $f(s_{kj})$ is a commit state
$\vdots$	$\vdots$
$<\cdots, s_{mi}, \cdots, s_{mj}, \cdots>$	$<\cdots, c_i, \cdots, c_j, \cdots>$

Since each global transition reflects one local state transition, two adjacent global states differ by exactly one local state. Therefore, either $s_{k-1,i} = s_{ki}$ or $s_{k-1,j} = s_{kj}$, and therefore, either $f(s_{k-1,i}) = f(s_{ki})$ or $f(s_{k-1,j}) = f(s_{kj})$. This implies that $f(s_{ki})$ or $f(s_{kj})$ is an abort state. By assumption, the other one is a commit state. Hence, F_k is inconsistent. □

Theorem 4: There exists no protocol resilient to a network partitioning when messages are lost.

Without loss of generality, we will restrict our attention to only two sites. Hence, the partitioning must be simple.

Sketch of Proof: Let $f(s)$ represent the result of a *timeout* transition from state s. Since messages are lost when the partitions occur, we can ignore the message state portion of a global state. Let $G_0, \cdots, G_m$ be a partition-free execution of the protocol that commits the transaction. Define F_i to be the global state resulting from a network partition occurring in state G_i. Using the notation of the previous theorem, $F_i = <f(s_{i1}), f(s_{i2})>$. As above, we can find the smallest k such that F_k contains a commit state. (Recall that F_0 contains only abort states.) Now, the difference between F_{k-1} and F_k is one local state. Therefore, F_k must contain an abort state as well, which makes the state inconsistent. □

Theorem 8: There exists no protocol resilient to a multiple partition.

In the proof of this theorem we will only consider protocols in which each state transition reads at most one message (however, a transition can still send an arbitrary number of messages). It is shown in [12] that these protocols are equivalent in power to more general protocols reading an arbitrary number of messages per state change. This assumption allows a simpler proof. The proof follows the same form as the previous nonexistence proof.

Proof: Let P be a three-site protocol that is correct in the absence of failures. We will assume that P is resilient to multiple partitions. Let $G_0, \cdots, G_m$ be a failure free path in the global state graph for P that commits the transaction. Now, $G_i = (S_i, M_i)$, where $S_i = <s_{i1}, s_{i2}, s_{i3}>$ is the vector of local states and M_i is the outstanding messages. We will consider M_i to be the union of three sets: M_{i1}, M_{i2}, and M_{i3}, where M_{ij} is the set of messages addressed to Site j.

Let G_i' be the global state resulting from a partitioning occurring during the global state G_i. Without loss of generality, we assume that all outstanding messages are returned to their senders. The recipients will receive only "timeouts." Let M_i' be the resulting set of messages. Thus, $G_i' = (S_i, M_i')$. Now, let f_j denote the transition function that moves Site j to a final state during a partitioning, i.e., $f_j(s_{ij}, M_{ij}')$ is Site j's resulting final state when a partitioning occurs during G_i.

Let k be the smallest k such that a multiple partitioning occurring while the transaction is in G_k still results in the transaction being committed. Since we have assumed that the protocol is resilient to such a partitioning, we have $f_j(s_{kj}, M_{kj}')$ equals commit for $j = 1, 2, 3$. Moreover, by our choice of k, we have $f_j(s_{k-1,j}, M_{k-1,j}')$ equals abort for $j = 1, 2, 3$. Now from G_{k-1} to G_k one state transition occurred, and let us assume that Site 1 made that transition. Furthermore, Site 1 read at most one message and, and if so, let this be a message from Site 2.

Notice that we have $s_{k-1,3} = s_{k,3}$ and $M_{k-1,3}' = M_{k,3}'$ since, between G_{k-1} and G_k, Site 3 did not make a transition and none of its messages were read. Therefore, $f_3(s_{k-1,3}, M_{k-1,3}') = f_3(s_{k,3}, M_{k,3}')$. But this is a contradiction. □

Acknowledgment

The authors wish to thank K. Birman, K. Keller, and L. Rowe for their useful comments and for valuable discussions.

References

[1] A. V. Aho, J. D. Ullman, and M. Yannakakis, "Modeling communications protocols by automata," in *Proc. 20th Annu. Symp. Foundations Comput. Sci.*, Oct. 29-31, 1979, pp. 267-273.

[2] P. Alsberg and J. Day, "A principle for resilient sharing of distributed resources," in *Proc. 2nd Int. Conf. Software Eng.*, San Francisco, CA, Oct. 1976.

[3] G. V. Bochmann, "Finite state description of communication protocols," *Comput. Networks*, vol. 2, pp. 361-372, Oct. 1977.

[4] J. N. Gray, "Notes on database operating systems," in *Operating Systems: An Advanced Course*. New York: Springer-Verlag, 1979.

[5] M. Hammer and D. Shipman, "Reliability mechanisms for SDD-1: A system for distributed databases," *ACM Trans. Database Syst.*, vol. 5, pp. 431-466, Dec. 1980.

[6] B. Lampson and H. Sturgis, "Crash recovery in a distributed storage system," Comput. Sci. Lab., Xerox Parc, Palo Alto, CA, Tech. Rep., 1976.

[7] B. G. Lindsay *et al.*, "Notes on distributed databases," IBM Res. Rep. RJ2571, July 1979.

[8] R. Lorie, "Physical integrity in a large segmented data base," *ACM Trans. Database Syst.*, vol. 2, Mar. 1977.

[9] D. A. Menasce and R. R. Muntz, "Locking and deadlock detec-

tion in distributed databases," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 195-202, May 1979.
[10] P. M. Merlin, "A methodology for the design and implementation of communication protocols," *IEEE Trans. Commun.*, vol. COM-24, pp. 614-621, 1976.
[10] J. B. Rothnie, Jr. and N. Goodman, "A survey of research and development in distributed database management," in *Proc. IEEE 3rd Int. Conf. Very Large Databases*, 1977.
[11] D. Skeen, "Nonblocking commit protocols," presented at the SIGMOD Int. Conf. Management of Data, Ann Arbor, MI, 1981.
[12] —, "Crash recovery in a distributed database management system," Ph.D. dissertation, Dep. Elec. Eng. Comput. Sci., Univ. Calif., Berkeley, 1982.
[13] M. Stonebraker, "Concurrency control and consistency of multiple copies in distributed INGRES," *IEEE Trans. Software Eng.*, vol. SE-5, May 1979.
[14] R. Schapiro and R. Millstein, "Failure recovery in a distributed database system," in *Proc. 1978 COMPCON Conf.*, Sept. 1978.
[15] L. Svobodova, "Reliability issues in distributed information processing systems," in *Proc. 9th IEEE Fault Tolerant Comput. Conf.*, Madison, WI, June 1979.

Dale Skeen was born in Concord, NC. He received the B.S. degree in computer science from North Carolina State University, Raleigh, and the Ph.D. degree in computer science from the University of California, Berkeley, in 1982.

He is currently an Assistant Professor in the Department of Computer Science, Cornell University, Ithaca, NY, a position he has held since 1981. His current research interests include distributed databases, highly fault-tolerant distributed systems, parallel algorithms for CAD/VLSI.

Dr. Skeen is a member of the Association for Computing Machinery, the IEEE Computer Society, and Phi Kappa Phi.

Michael Stonebraker, photograph and biography unavailable at the time of publication.

Dynamic Rematerialization: Processing Distributed Queries Using Redundant Data

EUGENE WONG, FELLOW, IEEE

Abstract—**In this paper an approach to processing distributed queries that makes explicit use of redundant data is proposed. The basic idea is to focus on the dynamics of materialization, defined as the collection of data and partial results available for processing at any given time, as query processing proceeds. In this framework the role of data redundancy in maximizing parallelism and minimizing data movement is clarified. What results is not only the discovery of new algorithms but an improved framework for their evaluation.**

Index Terms—**Distributed databases, distributed query processing, query processing.**

I. Introduction

IN THIS PAPER we propose a new formulation for the problem of processing queries in a distributed database system. By such a system we mean a collection of autonomous processors, communicating via a general communication medium, and accessing separate and possibly overlapping fragments of a database. The user's view of data is to be an integrated whole, both fragmentation and redundancy being invisible. Geographical dispersion, although sometimes present, is not an essential ingredient of such a system, and the range of systems so encompassed includes not only the classical geographically distributed databases but also configurations that are in effect database machines. The problem of distributed execution of queries is common to all these systems.

In the query processing algorithm designed for the SDD-1 distributed database management system [6], an irredundant subset of the database is used during the execution of any single query. No effort was made to exploit the possible existence of multiple copies either to maximize parallel operations or to minimize data moves. A related and somewhat hidden characteristic inherent in the SDD-1 algorithm is that parallel processing is opportunistic rather than deliberate.

These characteristics were recognized in [2] where the emphasis fell heavily on maximizing parallelism. The algorithm proposed there, and implemented for the distributed version of INGRES, achieves a high degree of parallelism by partitioning one relation among the processing sites and replicating all other needed relations at every site. We shall call this the FR (fragment and replicate) algorithm. For a query referencing many relations, the degree of data replication and the resulting communication cost to achieve this replication may be prohibitive. Thus, the FR algorithm is best applied to pieces of a many-variable query, one at a time, each with only two or three variables. Experience of using the FR algorithm in the

Manuscript received November 11, 1980; revised October 6, 1981. This work was supported by the Corporate Computer Science Center, Honeywell Corporation, the U.S. Army Research Office under Grant DAAG29-79-C-0182, and the U.S. Air Force Office of Scientific Research under Grant 78-3596.

The author is with the Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California, Berkeley, CA 94720.

0098/5589/83/0500-0228$01.00 © 1983 IEEE