A Locking Protocol for Resource Coordination in Distributed Databases

DANIEL A. MENASCE, GERALD J. POPEK, and RICHARD R. MUNTZ
University of California at Los Angeles

A locking protocol to coordinate access to a distributed database and to maintain system consistency throughout normal and abnormal conditions is presented. The proposed protocol is robust in the face of crashes of any participating site, as well as communication failures. Recovery from any number of failures during normal operation or any of the recovery stages is supported. Recovery is done in such a way that maximum forward progress is achieved by the recovery procedures. Integration of virtually any locking discipline including predicate lock methods is permitted by this protocol. The locking algorithm operates, and operates correctly, when the network is partitioned, either intentionally or by failure of communication lines. Each partition is able to continue with work local to it, and operation merges gracefully when the partitions are reconnected.

A subroutine of the protocol, that assures reliable communication among sites, is shown to have better performance than two-phase commit methods. For many topologies of interest, the delay introduced by the overall protocol is not a direct function of the size of the network. The communications cost is shown to grow in a relatively slow, linear fashion with the number of sites participating in the transaction. An informal proof of the correctness of the algorithm is also presented in this paper.

The algorithm has as its core a centralized locking protocol with distributed recovery procedures. A centralized controller with local appendages at each site coordinates all resource control, with requests initiated by application programs at any site. However, no site experiences undue load. Recovery is broken down into three disjoint mechanisms: for single node recovery, merge of partitions, and reconstruction of the centralized controller and tables. The disjointness of the mechanisms contributes to comprehensibility and ease of proof.

The paper concludes with a proposal for an extension aimed at optimizing operation of the algorithm to adapt to highly skewed distributions of activity. The extension applies nicely to interconnected computer networks.

Key Words and Phrases: concurrency, crash recovery, distributed databases, locking protocol, consistency
CR Categories: 4.33, 4.35, 5.24

1. INTRODUCTION

This paper is concerned with issues of resource coordination in distributed databases, and the maintenance of system consistency throughout normal and abnormal conditions. A database is said to be in a consistent state if all the data

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-77-C-0211. The work of D.A. Menasce was supported in part by the Conselho Nacional de Desenvolvimento Cientifico e Tecnologico, CNPq, Brazil.
A version of this paper was presented at the 1978 SIGMOD Conference.

Authors' address: Department of Computer Science, University of California, Los Angeles, CA 90024.

items satisfy a set of established *assertions* or *consistency constraints*. A database subject to multiple access requires that accesses to it be properly coordinated in order to preserve consistency. Coordination of resources in a distributed environment exhibits additional complexity over resource coordination in centralized environments due to

(1) Possibility of crashes of participating sites and or communication links. The occurrence of such failures can render the database inconsistent if not appropriately handled by the coordination algorithm.
(2) Network partitioning: In general, it is not possible to distinguish between messages which could not be delivered due to a crash of the recipient site and undelivered messages due to network partitioning. Therefore, network partitioning in the more general sense considered here is not simply a matter of proper network topology design. It turns out that detection of network partitioning can only occur at network reconnection time.
(3) Inherent communication delay: The time to get a message through a computer communication network may be arbitrarily long, although finite. Therefore, any proposed solution should operate correctly regardless of the delay experienced by any message, and in general should be efficient.

A protocol to coordinate concurrent access to a distributed database using locking is presented in this paper. The algorithm has as its core a centralized locking protocol with distributed recovery procedures. A centralized controller with local appendages at each site coordinates all resource control, with requests initiated by application programs at any site. Recovery is broken down into three disjoint mechanisms: for single node recovery, merge of partitions, and reconstruction of the centralized controller and tables.

Among the properties of the proposed protocol we have the following.

(a) *Robustness* in the face of crashes of any participating site, as well as communication failures, is provided. The protocol can recover from any number of failures which occur either during normal operation or during any of the three recovery processes. Recovery is done in such a way that maximum forward progress is achieved.
(b) *Deadlock prevention and/or detection methods* can be easily integrated given the centralized control characteristic of the proposed algorithm.
(c) *Straightforward integration of predicate locking methods* [6] is permitted. Value-dependent lock specification at the logical level is necessary to avoid the problems of "phantom tuples" discussed by Eswaran et al. [6]. Other locking disciplines may also be easily supported.
(d) *Continued local operation in the face of network partitioning is supported.* The locking algorithm operates, and operates correctly, when the network is partitioned, either intentionally or by failure of communication lines. Each partition is able to continue with work local to it, and operation merges gracefully when the partitions are reconnected.
(e) *Performance of the algorithm does not degrade operations.* It is shown in this paper that for many topologies of interest, the delay introduced by the protocol is not a direct function of the size of the network. The communication

cost is shown to grow in a relatively slow, linear fashion with the number of sites participating in a given transaction.

(f) The *correct operation of the protocol* in the face of the failures mentioned before can be proven in a straightforward way.

Several other approaches for synchronization in distributed databases have been suggested in the literature, but none deal satisfactorily with all of these issues.

The protocol presented in this paper is described in an intuitive manner in Section 2, followed by a more detailed description in the two subsequent sections. An algorithmic specification of this locking protocol can be found in [10]. An informal proof of the correctness of the algorithm is presented here. The proof is decomposed into five major parts, one for normal operation, three for the recovery phases, and a last part that shows the parts actually can be proved disjointly.

The paper concludes with a proposal for an extension aimed at optimizing operation of the algorithm to adapt to highly skewed distributions of activity. The extension applies nicely to interconnected computer networks.

1.1 Related Work

The majority consensus protocol proposed by Thomas [12] requires the sites involved in a transaction to agree by majority vote for it to proceed. Timestamps on data items at each site indicate whether the item is current and therefore whether a transaction based on it can be approved.

This protocol is quite elegant, with attractive behavior in the face of failures, especially for fully replicated databases. Unfortunately, for the cases considered in this paper, it presents several drawbacks. The locking discipline and scheduling of transactions are fixed by the nature of the algorithm itself, limiting flexibility (predicate locking cannot be supported for example). Performance can degrade severely with increasing system load in a thrashinglike manner, since several partially complete transactions which conflict lead to multiple resubmission of each. Partitions can render the protocol inoperative.

Synchronization in SDD-1 [2] is handled by several different protocols designed to coexist with one another. The simpler ones can be used for certain restricted classes of transactions known in advance of system generation. In such cases significant improvements in cost and delay over more general protocols result. Otherwise, however, we recommend our protocol since its performance is absolutely better and issues such as robustness and crash recovery, not handled by SDD-1, are considered fully.

A ring-structured solution is proposed by Ellis [5]. It uses sequential propagation of synchronization and update messages along a statically determined circular ordering of the nodes. Two round trips are required for each update. This protocol, while in general much slower than the others mentioned above, is quite simple and Ellis has employed formal verification procedures to show its correctness. However, failures and error recovery are not addressed by the protocol.

Other proposed schemes, called *primary copy* strategies, have been suggested in [1, 3, 7]. Alsberg et al. in [1] introduced some techniques aimed at providing a certain degree of resiliency to the single primary, multiple backup strategies discussed in [3] and [7]. The primary copy scheme is primarily designed to maintain mutual consistency of databases subject to somewhat limited types of update operations, but it does not explicitly address the problem of internal consistency of a distributed system supporting general transactions.

2. CENTRALIZED LOCK CONTROLLER PROTOCOL—INTUITIVE DESCRIPTION

The database we are considering here is distributed among n nodes of a computer network, numbered from 1 to n. We assume that the network protocols are such that a copy of a message is kept by its sender until an acknowledgment for it is received. In other words, there are no lost messages. The only exception is messages discarded due to crashes of nodes and communication outages. However, messages may have to be retransmitted many times until they get through the net. An implication of these assumptions is that messages may be delayed by an arbitrary but finite amount of time. We also assume that messages from a source site A are delivered by the network protocols to a destination site B in the same order that they were generated. However, we make no assumptions about the order in which messages from two distinct sources are received by a third one. We require that the network routing procedures be such that every pair of nodes can communicate with each other if the necessary physical connection is available. It is also assumed that crashes are detectable, rather than just returning incorrect data.

User interaction with the database is done through application programs, APs, which communicate with the database management processes. Of those processes, two are of interest for this locking protocol: the *centralized lock controller* or simply lock controller and the *local lock controller.*

As a first approximation assume that there is only one lock controller or LC for the entire network. This process is responsible among other things for examining lock and lock release requests from the APs, and deciding whether they should be granted or not. For this purpose, the LC maintains a table called the LOCK table, which is a set of all the active locks. Each entry in this table is a 3-tuple of the form $\langle H, T, P \rangle$ where H is a unique host identification, T is a unique database transaction identifier within each site, and P is a description of the logical portion of the database to be locked as well as the lock mode (e.g., read, write, etc.). In a relational database, the lock specification may, for example, be a predicate lock as described by Eswaran et al. [6]. Note that the pair $\langle H, T \rangle$ is a globally unique transaction identifier.

At every site, except for the one where the LC is located, there is a local lock controller or LLC. Those processes are responsible for maintaining a local copy of the relevant portion of the LOCK table. The relevant portion of the LOCK table is the set of entries which contain locks which refer to data stored locally. Any LLC may become the lock controller whenever there is a crash in the system which makes the LC unavailable. The recovery process is explained later in detail. Each time a transaction takes an action the local copy of the LOCK table is examined to determine whether the action can be performed or not. Therefore, there are two reasons for keeping a local copy of the LOCK table, namely: resilience to failures and local action checking.

It is convenient at this point to introduce the notion of logical partition or logical component, as opposed to that of a physical component. A *physical* *component* is a maximal subnetwork such that every pair of sites in the component can communicate with one another. It can be readily seen that the composition of a physical component is not under the control of the locking protocol, since nodes and communication links fail independently of the protocol operation. Such a lack of control could make the operation of the protocol, in the face of crashes, rather complex. The concept of logical component is introduced to give the protocol independence from unexpected changes in the composition of each physical component. To this end, each LC keeps a list of sites which he thinks are still up, called the up list. A *logical component* is defined as being the subnetwork indicated by the nodes which are in the up list. This list may lag behind the list of sites which are actually up. Independence from the composition of physical components is thus achieved by controlling the way by which the latter list maps into the former, in a way which is explained later in the paper.

Since one of our stated goals is to allow local operations to continue in the face of network partitioning and to allow partitions to merge gracefully, it is necessary for each partition to have its own LC. There is one LC for each logical component.

The operation of the locking protocol under no crash conditions can be intuitively explained as follows. The LC receives lock and lock release requests from the application programs. The LC then assigns a sequence number to the request. These sequence numbers are taken from a monotonically increasing sequence of numbers and will be used for the purposes of crash recovery, as will become clear in a later section. Each request (including its sequence number) is then sent to all *relevant* LLCs in the component. An LLC is relevant with respect to a request if its site stores data addressed by the request in question. The request is stored in a pending list at each LLC site and an acknowledgment is sent back to the LC. After the acknowledgment from all relevant sites in the component is received (excluding those which crashed in the meantime) a confirmation for the request is sent by the LC to all LLCs causing the request to be deleted from the pending list and appended to the LOCK table.

An LC may choose to reject a lock request if it conflicts with other locks in the LOCK table or in the pending list or if the request is not *local* to the component.[1] We assume that the LC is able to determine for each lock, P, the set, LOC(P), of sites where the data to be locked are stored. Thus a lock P is said to be local if LOC(P) is contained in the up list for the component. The set LOC(P) can be determined by the LC by checking some catalogs. The organization of those catalogs is not relevent here; see [4] and [11] for discussions of that subject.

Every time that a site or a set of sites drops out of the up list, all the transactions which had at least one lock which became not local will be aborted or backed up. All the locks held by these transactions are released. In this way complete locality of operations is enforced by the CLC protocol.

If the LC crashes or becomes unavailable a recovery mechanism called *logical component recovery* (*LCR*) takes place. As soon as an LC-crash is detected by any process engaged in a conversation or exchange of messages with the lock controller, a new process is nominated to be the new LC. There is a globally

[1] Reading of replicated data can easily be supported by whatever decision procedure is chosen so long as at least one copy is local, and update in other components is by convention blocked. If update in general is desired, then undo/redo facilities such as those outlined by Gray [8] would have to be added.

known circular ordering of the sites from which the nominee is selected. If the nominee is up it accepts the nomination by sending a message which circulates through all the sites in the component. The purpose of this message is also to collect all the requests which have been received by all the relevant sites but which are still in the pending list for at least one of these sites. Those requests will be incorporated into the LOCK table at every site in a subsequent phase of the recovery process. In summary, the LCR mechanism amounts to electing a new LC for the component and updating all the LOCK tables appropriately before normal operation is resumed. Various race conditions are dealt with by the details of the recovery protocol.

It is the responsibility of each LC to periodically monitor the connection between it and a node not in its up list. If a physical connection between two previously logically disconnected components is detected, a *logical component merge* (*LCM*) mechanism is started. LCM is always done pairwise between components and in this process the LC of one of the components plays an active role while the other plays a passive one. The first phase of LCM is composed of an interconnection protocol by which two LCs are logically connected in such a way that one of them is designated active and the other passive. This protocol also enforces the pairwise merge condition and is shown to be deadlock-free. After a logical connection has been established both LCs clear all outstanding requests and reject further ones. In the subsequent phase, the union of the LOCK tables of the two components is made and the new LOCK table is sent to all the sites in both components in the form of a message which circulates through them. This message signals the completion of the merge. The active LC becomes the lock controller for the new logical component.

When a site which was down recovers, it is made active by the *single node recovery* (*SNR*) mechanism which basically amounts to the acquisition by that site of a new copy of the relevant portion of the LOCK table.

The three recovery mechanisms described above do not interact with each other, as will be shown later. This property is important because it allows us to decompose the correctness proofs into a proof of disjointness and then proofs for each recovery procedure separately.

The recovery mechanisms will be shown to be robust in the face of additional failures. In order to achieve this goal, each mechanism is designed in such a way that a partial execution of any of the recovery algorithms does not destroy any of the properties we want to prove about them.

It is important to emphasize at this point that, since all the lock requests are examined by a centralized lock controller in one logical component, locks granted by an LC do not conflict with one another. This fact enables us to consider the operation of the algorithm for normal operation and for recovery as if there were only one lock per logical component. The reader is encouraged to keep this in mind as he reads through this paper.

3. LOCK AND RELEASE GRANTING ALGORITHMS

This section describes informally the algorithms used to grant new locks and to release existing ones. One would like those algorithms to have the property that a lock is either granted or released if and only if it is known to all the relevant sites. The basic structure of both algorithms can be abstracted in what we call the *SafeTalk protocol* (*STP*) which exhibits the desired property outlined below.

Let there be a sender S, who wishes to send a message M, originated at an external source ES, to n destinations D_1, D_2, . . . , D_n. Each site i keeps two message buffers: temp_buffer(i) and final_buffer(i). SafeTalk is such that message M will only be in final_buffer(S) if M is either in temp_buffer(D_i) or final_buffer(D_i) for all destinations D_i. SafeTalk can be described by the following set of rules:

(1) S receives a "MESSAGE REQUEST" or MR message from ES and broadcasts an "ACCEPT MESSAGE" or AM message, which contains M, to all D_i's, $i = 1, \ldots, n$. M is tagged by S with a sequence number to distinguish M from preceding or subsequent messages. Sequence numbers are monotonically increasing. We denote the tagged message as M^+. The message M^+ is placed in temp_buffer(S).
(2) When an AM message is received by a destination D_i, the message M^+ is placed in temp_buffer(D_i) and a "MESSAGE ACCEPTED" or MA message is sent back to S.
(3) When all the MA messages have been received by S, M^+ is moved to final_buffer(S), removed from temp_buffer(S), and a "CONFIRM MESSAGE" or CM message is broadcast to all destinations.
(4) The receipt of a CM message at destination D_i causes M^+ to be moved into final buffer(D_i) and removed from temp_buffer(D_i).

A variant to this protocol, called a two-phase commit protocol, is described in [8] and [9]. The two-phase commit protocol has an additional acknowledgment message at the end from each destination to the sender. However, we show here that this additional message is unnecessary. Therefore, the two-phase commit protocol has one-third more messages than the SafeTalk protocol. The SafeTalk protocol is illustrated in Figure 1. In this figure, the horizontal arrows are labeled with message names. There is one vertical line corresponding to the sender S and another corresponding to destination D_i. Two graphical notations were used in this figure. Namely, the diverging arrows shown at point A of the vertical axis for the sender S indicate that the message in question is being broadcast to every one of the destination sites. The converging arrows shown at point B of the same vertical axis indicate that the sender must wait until it receives all the messages from all the sites which are up. Previously up sites may be pronounced down by the underlying network protocols after timeout and retransmission take place several times.

Several other details are also worth keeping in mind. As mentioned before, each LC keeps a list of the sites in the component which are up. A node i is removed from this list by the LC each time that the underlying network protocols fail to deliver a message to site i (after timeout and retransmission occurred a certain number of times). An up list is also modified by the execution of any of the three recovery mechanisms. A copy of the up list is also kept by each LLC. Every update to the up list by the LC is transmitted to all LLCs in the component. Note that no additional message traffic is generated by those updates since they can "piggyback" on other messages. The reason for keeping local copies of the up

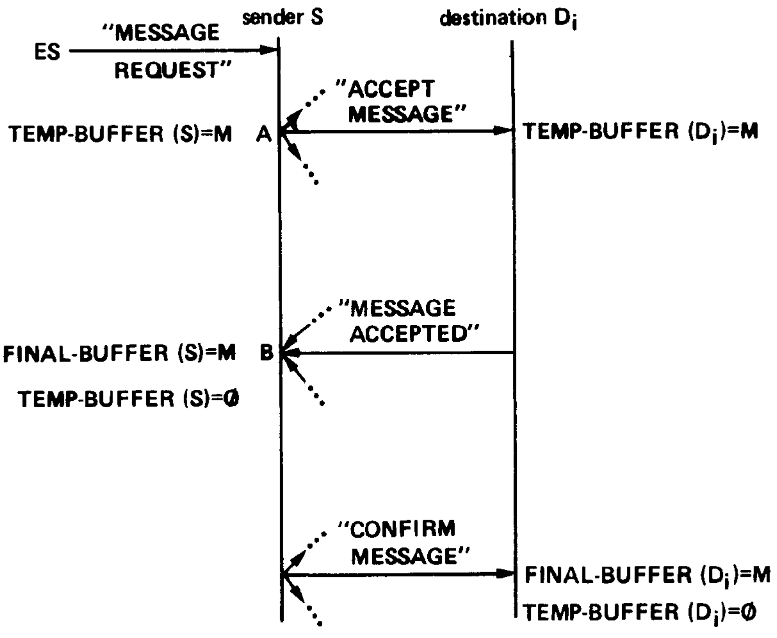


Fig. 1. SafeTalk protocol (STP).

list is partly a matter of performance, since the up list determines to some extent the set of nodes which should participate in the LCR or LCM recovery mechanisms, as will be seen later. Also, every time that a change in the up list causes certain locks not to be local anymore, all nonlocal locks are released and the affected transactions aborted.

3.1 Lock Granting Algorithm

Application programs issue lock requests by sending a "LOCK REQUEST" or LR message to the LC. This message contains the lock or 3-tuple which the user would like to be entered in the LOCK table. The LC decides whether the lock can be granted or not. If the requested lock conflicts with other active locks a scheduling decision must be taken by the LC as to whether to preempt any transaction or to make the requester wait. That decision is not the concern of this paper. If there are no conflicts and the lock is local to the component the LC must notify every relevant LLC in its component that a new entry should be appended to their LOCK tables. Actually, instead of inserting the lock directly into the LOCK table, an LLC appends it to a list of pending lock requests, called an *L-list*. The reason for this is to prevent copies of the LOCK table from becoming inconsistent if the LC crashes.

The basic structure of the lock granting and lock releasing algorithms is the same as that of the SafeTalk protocol, where AP, LC, LLC_i, and LOCK table correspond to ES, S, D_i, and final_buffer in SafeTalk, respectively. Also, the message M in SafeTalk should be considered as a lock request for the lock granting algorithm and as a release request for the lock releasing one. For the lock granting algorithm, in particular, temp_buffer corresponds to an L-list.

3.2 Lock Releasing Algorithm

A similar procedure is followed when an AP issues a lock release request, by sending to the LC a "RELEASE REQUEST" or RL message. Each site keeps a list of pending release requests or an *R-list* for the same reasons we introduced the L-list. The R-list corresponds to temp_buffer in the SafeTalk protocol.

The operation of the lock granting and lock releasing algorithms is described in a more formal way in Appendix A. This description is in the form of a pair of interacting state transition diagrams. The rules which determine the set of feasible global states for the system are presented in Appendix A. In Section 8, several of these logically distinct messages are physically combined to assure suitable performance. The next section gives some definitions and proofs regarding the global feasible states of the LOCK tables, L-lists, and R-lists.

3.3 Some Definitions and Proofs

We show here that if no crash occurs the lock granting and lock releasing algorithms have the property that a lock is only granted or released if all the sites in the component know about the request. In order to make this statement more precise consider the following definitions. Let LT_i, L_i, and R_i be the LOCK table, L-list, and R-list at site i, respectively.

Definition 1 (*Lock Request Presence*). A lock request x or a lock is said to be *present* at site i if $x \in [LT_i \cup L_i] - R_i$.

Definition 2 (*Release Request Presence*). Let x be a lock request and let y be its associated lock release request. It is said that y is *present* at site i if $y \in R_i$ or if $x \notin LT_i \cup L_i$.

It is convenient at this point to define precisely the meaning of a site being relevant to a lock. We say that site i is *relevant* to lock x if at least one of the data items addressed or covered by x are stored at site i. Let us define $S(x)$ as the set of sites relevant to the lock x. We can now make the following statements.

Assertion 1. If a lock $x \in LT(LC)$ and if there is no pending release request associated with x at the LC, then the lock request x is present at every site in $S(x)$.

Assertion 2. If a lock $x \notin LT(LC)$ and if there is no pending lock request associated with x at the LC, then the release lock request y associated with x is present at every site in $S(x)$.

The proof for these two assertions, as well as for all other assertions in this paper, can be found in Appendix B. Together, they lead directly to the following result.

THEOREM 1. *Let C be a logical component, LC its lock controller, and U the set of sites in C. If no crashes ever occur then a lock request is only granted by the LC after it is present at all the relevant sites in U and a lock is only released if the associated release request is present at every relevant site in U.*

In a later section the results in this section will be extended to deal with situations in which crashes can occur.

4. CRASH RECOVERY

So far we have described the protocol for requesting locks and releasing them, assuming that no crash occurred. Communication links, processors, operating systems, and processes are some examples of sources of crashes.

The three already mentioned recovery mechanisms are presented here. These mechanisms are proven to be robust with respect to additional failures. To be robust, the protocols must preserve logical component internal and mutual consistency as defined below, if any changes have been made to any permanent information (like LOCK tables, up lists, or LC id's) at any node.

Definition 3 (*LT-Consistency*). The set of LOCK tables of a logical component is said to be LT-consistent if Assertions 1 and 2 hold at any time.

Definition 4 (*Logical Component Internal Consistency*). A logical component is said to be internally consistent if the set of its LOCK tables is LT-consistent and if there is one and only one LC, whose identity is known to every node in the component.

Definition 5 (*Logical Component Mutual Consistency*). A set of logical components is said to be mutually consistent if all of them are internally consistent and if there is no lock present at any LOCK table of one of them which conflicts with another lock of any other component.

Definition 5 covers Definitions 3 and 4, and specifies an important property which is required of recovery.

The recovery protocols have been designed so that all crashes which can occur during a recovery phase fall into one of the two disjoint classes, which we call *terminal* and *transparent* failures.

A terminal crash causes the entire recovery mechanism to be aborted and restarted. The possible conditions under which terminal crashes occur are shown to leave the protocol in a robust state, as defined above. A transparent crash is defined to be one which does not affect the continued correct operation of the recovery process.

The following is the definition of robustness used in this paper. A recovery protocol is *robust* if all crashes can be shown to be either terminal or transparent. As we shall see, for each of the recovery mechanisms, we can identify a point before which the recovery can be considered as not having happened at all and after which it is considered to be successfully carried out. This point is called the *completion point.* Crashes before the completion point, if they have any effect at all, are shown to be terminal. Crashes after the completion point are shown to be transparent.

The three proposed recovery mechanisms are shown to occur disjointly in time. In other words, a merge of two logical components only takes place if both are in their normal state or are not recovering from a logical component crash. Also, a site only becomes attached to a logical component if this component is in its normal state. These important properties allow us to state and prove separate theorems concerning each one of them.

4.1 Logical Component Recovery (LCR)

We now show how an LLC may become an LC if the LC crashes. A crash of the LC can be detected by any process engaged in a conversation or exchange of messages with it. As an example, an AP may time out while waiting for a reply from the LC for a lock or lock release request. In every case, the process which detects a crashed LC is responsible for nominating a new LC. For this purpose, we assume that the distinct sites or nodes in the underlying network are arranged in a linear order such that node $\#i$ precedes node $\#(i + 1) \bmod n$. Let this order be called the *nomination order*. So whenever a process detects a failed LC it nominates the next node which is up in the nomination order after the crashed LC to the position of LC. This nomination is accomplished by the issue of an "ACCEPT NOMINATION" or AN message by the nominator. If this message is not acknowledged after a certain number of times it has been retransmitted, the nominator assumes that the nominee is down and sends an AN message to the next site in the nomination order. However, it may be the case that the originally nominated node was not down, as assumed by the nominator, but that due to certain conditions in the network its reply was seriously delayed. So it seems that more than one LC could be nominated in this process! Let us neglect this issue for the moment, while we describe the recovery procedure, and show later how such an undesirable situation can be easily avoided. The nominee is first responsible for checking that the old LC is actually dead (since the nomination may have come from an errant AP). Then the nominee must notify every other site that it has accepted the nomination. Moreover, the nominee must make sure that all the copies of the LOCK table are appropriately updated. From now on, we refer to the crashed LC as the "old LC" and to the nominee as the "new LC."

The process by which the new LC becomes the actual LC can be divided into two phases: a "notification phase" and a "LOCK table update phase."

In the notification phase all the nodes in the component, as indicated by the up list U, are informed of the identity of the new LC. Also, in this phase enough information is gathered in order to appropriately update the LOCK tables in the subsequent phase. The update of the LOCK tables is done in such a way that the *maximum forward progress* is obtained at the end of the LCR procedure. More precisely, in terms of the state tables (STDs) introduced in Appendix A, the LCR mechanism leaves the system in the global state which would have been reached by the system if a crash had not occurred and if no new requests were submitted.

The new LC, upon nomination, issues a message called "NOMINATION ACCEPTED." This message circulates once through the set of all sites in U (including the site where the new LC runs) in a predetermined order. Notice that the up list U of the nominated LC determines the logical component to be restored by the logical component recovery procedure.

During the NA cycle, two sets will be constructed, namely, the set L of locks to be added to all the LOCK tables and the set R of locks to be deleted from all the LOCK tables. The set L includes all the locks which are in at least one lock pending list and that would therefore end up in all LOCK tables if no crash had occurred. The set R includes all the locks which are in at least one lock release list and would therefore be deleted from all LOCK tables under normal conditions.

It is possible for a given lock to be in the L-list at one site and at the R-list at another site. This situation can be seen from Table I and is also illustrated by the following scenario. Consider an LC and two LLCs, LLC1 and LLC2. Consider a lock *x* which was accepted by LLC1 and LLC2 already. The LC then enters the lock into its LOCK table and sends the CONFIRM LOCK (CL) message to LLC1 and to LLC2. LLC1 receives the message and moves the lock into its LOCK table. Now, the LC receives a release request for the same lock. It then sends an ACCEPT RELEASE (AR) message to LLC1 and LLC2. LLC1 receives it and enters the request into its R-list. LLC2 did not receive either the CL or the AR messages so far and therefore the lock *x* is still in its L-list.

While constructing the sets L and R one has to be able to decide whether a lock should be installed in all the LOCK tables or deleted from all of them in those cases in which the request appears in both L-lists and R-lists. This decision is easily made with the aid of the sequence numbers which the LC attaches to every request. The greater the sequence number the later is the request. Therefore, the latest requests are considered when constructing the sets L and R.[2]

Every node in the NA cycle, other than the new LC, receives partially constructed sets L and R, adds its contributions to them, and places the new versions of the sets into the NA message which is forwarded to the next node in the cycle. When the NA message returns to the new LC, the sets L and R are completed. The sets L and R are modified at site *i* according to the algorithm given below.

```
for $x \in L_i$ do
  if there is $y \in R$ associated with $x$
  then if sequence $\#(x) >$ sequence $\#(y)$
    then begin
        comment lock request is the latest;
        $L := L \cup \{x\}$; $R := R - \{y\}$;
      end;
    else;
  else $L := L \cup \{x\}$;
for $y \in R_i$ do
  if there is $x \in L$ associated with $y$
  then if sequence $\#(y) >$ sequence $\#(x)$
    then begin
        comment the release request is the latest;
        $R := R \cup \{y\}$; $L := L - \{x\}$;
      end;
    else;
  else $R := R \cup \{y\}$;
```

The actual update of the LOCK tables is done using the SafeTalk protocol. Therefore, if an additional crash occurs during the LOCK table update phase, the set of possible states in which the LOCK tables, L-lists, and R-lists may be left is the same as the set of possible states which can result if the crash had occurred during normal operation. Therefore, recovery from additional failures merely requires restarting the LCR mechanism again.

After the notification phase is over, the new LC sends a message to every LLC

[2] The two-phase commit protocol disambiguates this case at the additional cost of the extra, fourth message.

asking them to update their LOCK tables. This message is called an "UPDATE TABLE" or UT message, and it carries within it the sets L and R. The actual set R contained in this message also includes lock release requests for transactions which are not local anymore. These transactions are first aborted and their locks released when the LOCK table is updated. Upon receipt of the UT message, the L-list of each site is made equal to the set of locks in the set L which are relevant to the site. Analogously, the R-list of each site is made equal to the set of locks in the set R which are relevant to the site. After this is done, each site sends a "READY TO UPDATE" message or RU message to the new LC.

After receiving an RU from every up site the new LC becomes the actual LC by notifying all the LLCs that they can resume their normal activity. For this purpose the LC broadcasts a "RESUME NORMAL ACTIVITY" or RNA message. The new value for U is the set of sites from which the LC received an RU message. This new value for U is included in the RNA message, thus allowing every node in U to know the composition of the set U. Also, upon receipt of the RNA message, the LOCK tables are updated accordingly to the L-lists and R-lists.

Let us now describe how we can guarantee that only one LC will emerge from the notification process. Recall that the nominator will nominate the first up node in the nomination sequence. Let us make the following definition.

Definition 6 (*Trial Sequence*, $T[j, k]$). A trial sequence, $T[j, k]$, is the sequence $i_1, i_2, \ldots, i_{k-1}$ of site numbers for which an "ACCEPT NOMINATION" message has been unsuccessfully sent by a nominator j, before j sent an AN message to site #k.

For every AN message sent from site #j to site #k we include the sequence $T[j, k]$ as part of it. This sequence is also included as part of the "NOMINATION ACCEPTED" message which circulates through the set of sites. The purpose of this is to allow any site to resolve any conflict that can arise due to the race conditions discussed earlier in the paper. Namely, it is possible that more than one LC was nominated and consequently more than one NA message (from distinct sources) would be circulating. Conflicts are resolved by giving preference to the last LC to be nominated. NA messages originated by other nominated LCs are killed when they are detected to belong to the improper LC. Therefore, each time an NA message is received by node i the algorithm, shown below in Algol-like notation, is executed.

ALGORITHM TO PROCESS AN NA MESSAGE AT SITE i

Assume that in each site there is a *trial list* T, which is empty if no NA message has already been received in the present recovery phase and that is updated by the algorithm below (as executed at site i). Let the variable new_LC$_i$ be the identification of the new LC site as imagined by node i. Let $T[j, k]$ be the trial sequence included in the NA message.

```
if $T$ is empty or $k \notin T$
then begin
  comment either no NA message has already been received for this recovery
```

```
            process or the current NA message belongs to a more recently
            nominated LC;
    $T := T[j, k]$;new_LC$_i$ := $k$;
    **end**;
**else** Send "KILL NA" message to site $k$ informing it that its cycle was aborted;
```

In many instances in this protocol we require a certain message to circulate through a set of nodes, as is the case with the NA message. Let us call such messages "circular messages." They always have a source or generator that is responsible for sending it through a cycle. The underlying network protocols assure us that messages will not get lost while going from one site to another by the use of timeout and retransmission schemes. However, a circular message can still be lost if a node in the cycle crashes after receiving it but before being able to forward it. The loss of a circular message can be prevented by having each node in the cycle send to the circular message generator an acknowledgment for it, but only after it was forwarded to the next node in the sequence. Now the source is able to detect a cycle interruption and it can appropriately resume it by sending the last copy of the message to the appropriate site.[3] This source acknowledgment scheme at the centralized lock controller protocol level is assumed to exist whenever a circular message is necessary.

It should be noted that if an application program issues a lock or release request and the LC fails before the request is present at every site, the request will never appear in the local LOCK table even after the LCR is completed. Therefore, APs should timeout for requests and resubmit them.

The LCR mechanism can be described in an abstract form by the STDs in Figure 2. Figure 2(a) is the STD for the new LC and Figure 2(b) is the STD for any local lock controller participating in LCR. Let LC be the identification of the old LC and let LT be its LOCK table. Let LC* be the new LC and LT* be its LOCK table. Associated with each state in the STDs of Figure 2 is a 3-tuple of the form (a, b, c) where a, b, and c are the LC identification, the new LC identification, and the LOCK table, respectively.

4.2 Proofs About LCR

We would now like to prove that the notification phase ends with one and only one LC having been successfully nominated, and that all sites know the correct new LC identification. As a first step we state Assertions 3 and 4 which are concerned with the behavior of LCR given that no additional crashes occur.

Assertion 3. Given that no additional crashes occur during LCR, there will be one and only one LC whose identification is known to all sites in the component at the end of the notification phase.

Next, let a *globally accepted* lock (release) request be one which is in all L-lists (R-lists) of a logical component.

Assertion 4 (*Maximum Forward Progress Assertion—No Additional Crashes*). Given that no additional crash occurs, the following is true at the end

[3] This procedure can be optimized to a desired level by having only every ith node send an acknowledgment back to the generator of the circular message. Then there is additional uncertainty over which node in the cycle failed. It is up to the generator to resolve the uncertainty.

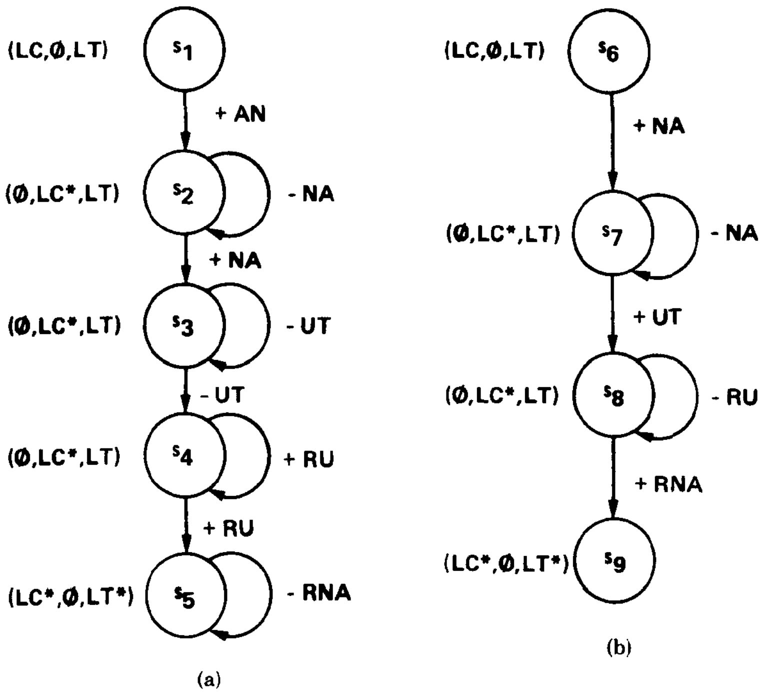


Fig. 2. STDs for LCR.

of the LCR mechanism. A lock p is in the LOCK table of all sites of $S(p)$ if the lock would have been in the LOCK table of the crashed LC if the crash had not occurred and if no new requests were submitted. Otherwise the lock is in none of the LOCK tables at the sites of $S(p)$.

Given these assertions we prove the robustness of the LCR mechanism.

THEOREM 2. *The logical component recovery (LCR) algorithm is robust.*

PROOF. The completion point for this algorithm occurs when the new LC has logged the fact that it is the LC after at least one RNA message has been sent. This point is indicated as state s_5 in the STD of Figure 2(a). The only terminal crash is a new LC failure before this point. This crash when detected will cause another LC to be nominated and the LCR mechanism to be restarted. In order to prove that LCR is robust we must prove that internal consistency is not violated by the partial execution of LCR. In other words, we need to show that the set of LOCK tables of the component is LT-consistent. The crash of the new LC can occur:

(a) before any LOCK table has been updated;
(b) after some but not all LOCK tables have been updated;
(c) after all LOCK tables have been updated.

In case (a) it is clear that the partially executed LCR has no effect at all. In case (c) all LOCK tables will be identical; therefore internal consistency for the component in question is trivially satisfied. In case (b) the crash occurred in the middle of the LOCK table update phase. But since LOCK tables are updated using the SafeTalk protocol, the states of the LOCK tables, L-lists, and R-lists at the instant of the crash is a state which could have resulted if the crash had occurred during normal operation. Therefore, the partial execution of the LCR mechanism does not violate the internal consistency of the component. The LCR mechanism will be restarted again as many times as crashes occur.

Now it remains for us to analyze the transparent failures. These are all the failures other than the new_LC crash already discussed. We can have either a process or processor failure which simply knocks out one of the sites in the component, or the component can be partitioned into two or more components. In either case, a set of one or more nodes is isolated from the set of nodes which participates in the LCR mechanism. The nodes in this set will not be considered any more for the rest of the LCR algorithm. However, we have to show that no inconsistencies are generated by a node dropping out during the execution of LCR.

For this purpose, we examine all the possible instants at which a node j may crash.

Case 1. During the "nomination phase." Here we have to show that the sets L and R will not be perturbed by any contributions already made to them by node j. Node j can crash at three possible instants.

(a) Before the NA message first reaches it. In this case node j is simply removed from the up list U without contributing to the formation of either L or R.
(b) After the NA message reaches it and before it is forwarded to the next node in the sequence. Here, the node which sent the NA message to node j will timeout, detect its crash, and send the NA message to the node which follows node j in the sequence. Again no contributions have been made to the sets L or R.
(c) After the NA message has been forwarded. A crash of node j at this point is equivalent to a crash of a node during the "LOCK table update" phase since node j already played its role in the "notification phase." Therefore, this case reduces to the next one to be examined. The reader should notice that the robustness of this recovery mechanism relies heavily on the fact that the SafeTalk protocol is used in the LOCK table update phase.[4]

Case 2. During the "LOCK table update phase." A crash of a node during this phase will have no effect upon other nodes, resulting only in the removal of this node from the up list of the logical component which is recovering.

Examination of all these cases completes this proof. □

The above result allows us to relax the assumption made in Assertion 4 that no additional crashes occur during LCR and state the following assertion.

Assertion 5 (*Maximum Forward Progress—Additional Failures Allowed*). Let C be a logical component. Let U be the up list which defines the component C. Let U′ be a subset of U which indicates the nodes which are

[4] Note that U = U′ if any node crashes before the end of LCR but after all the RU messages have been received.

actually up at the end of the LCR mechanism. The following is true at the end of the LCR mechanism. A lock p is in the LOCK table of all sites of $S(p) \cap U'$ if the lock would have been in the LOCK table of the crashed LC if the crash had not occurred and if no new requests were submitted. Otherwise, the lock is in none of the LOCK tables of the sites of $S(p) \cap U'$.

Finally we prove that every logical component is internally consistent.

THEOREM 3. *Every logical component is internally consistent.*

PROOF. Let C be any logical component. We have to prove that

(a) The set of LOCK tables of C is LT-consistent.
(b) There is one and only one LC for C.

Statement (a) is clearly true for normal operation of component C since Assertions 1 and 2 were demonstrated for this case. Now, by Assertion 5 either a lock p is in all the LOCK tables of $S(p)$ or it is in none of them at the end of LCR. Therefore, Assertions 1 and 2 are trivially satisfied and consequently LT-consistency is preserved.

Statement (b) was proved to be correct in Assertion 3 for the case in which no additional crashes occur during LCR. But, by Theorem 2, LCR is robust. This allows us to consider the effect of LCR as if no additional crashes occur during its execution, and concludes the proof. □

4.3 Single Node Recovery

So far we have described how the system recovers from a logical component crash. We now show how a node which is down becomes active again, or in other words, how it gets logically connected to a logical component. Let node j be such a node. The first step to become active is to find out the identity of any LC. This step is carried out by sending the "WHO IS THE LC?" or WLC message to any up node. Assume first that node j received at least one reply to its WLC message containing the identity of an LC. In this case, node j sends a message called "Hı THERE" or HT to the LC telling him that node j is alive again. If the LC is not undergoing any kind of crash recovery it will send to node j the portion of its LOCK table relevant to node j as well as its up list. An "ACCEPT LOCK" or "ACCEPT RELEASE" message is sent to node j by the LC for every lock or release lock request for which not all the LA or RA messages have been received.

Assume now that node j does not get any answer or that all the nodes which replied to its WLC message are themselves recovering from a crash or coming up from normal system shutdown. Then, node j becomes a logical component on its own. Node j is the LC for this logical component and the LOCK table, L-list, and R-list are initialized as empty. The logical component merge procedure described in Section 4.5 will take care of integrating node j into another logical component.

4.4 Robustness of SNR

THEOREM 4. *The single node recovery (SNR) algorithm is robust.*

PROOF. The only case of interest is the one in which the recovering node is able to get the identity of an LC as a response to its WLC message since, otherwise the SNR mechanism degenerates into an LCM as already explained above. Let j be the recovering node and let LC_i be the LC to which node j is trying to connect. The proof is extremely simple since the only two crashes of interest are (a) LLC_j crash and (b) LC_i crash. Case (a) is clearly a terminal case. Case (b) is also a terminal crash since a crash of LC_i, before it is able to send the LOCK table to LLC_j, prevents the LOCK table from being received by node j, thereby implying that SNR must be restarted. This completes the proof. □

4.5 Logical Component Merge

As a result of the logical component recovery algorithm an LC will be elected in each logical component of the network. Transactions which are local to a component will continue to be serviced as if no disconnecting crash had occurred. On the other hand, transactions which span more than one component will have to wait until the components involved are brought together again. It is the responsibility of each LC to detect when two components are physically connected again and to take the necessary steps to merge them into one logical component. The merge of logical components will always be done on a pairwise basis. The whole logical component merge mechanism is divided into two phases, namely a "reconnection detection" phase and a "merge" phase.

In the reconnection detection phase, each LC sends periodically a "WERE YOU ALIVE" or WYA message to every node not in its up list. The purpose of this message is to detect the existence of sites which were not reachable before but which were up. For the purposes of the description that follows, let the two logical components to be merged be called C1 and C2. Let LC1 and LC2 be their respective LCs and U1 and U2 their respective up lists. LC1 will take an active role during the whole recovery phase, while LC2 will take a passive one. As we shall see, a crash of LC1 while the recovery mechanism is in progress will result in abort, while a crash of LC2 after the reconnection detection phase is tolerated. Assume now that site #j in C2 received a WYA message from LC1. A component is said to be in NORMAL status if it is not undergoing any kind of crash recovery mechanism. If component C2 is in its NORMAL status, site #j sends a "YES I WAS" or YIW message to LC1. This message carries within it the identification of LC2.

At this point LC1 has to establish a logical connection with LC2. This connection is called a primary-secondary or P-S connection type with LC1 being the primary and LC2 the secondary. Since we require that LCM be done in a pairwise basis, the following conditions must be enforced by the protocol that establishes a P-S connection:

C1. An LC cannot be primary (secondary) for more than one P-S connection.

C2. An LC cannot be primary and secondary simultaneously.

The P-S connection is attempted by having LC1 send a "LET US MERGE" or LUM message to LC2. This message contains the up list U1. The status of LC1 is now changed to ATTEMPT. If the status of LC2 is NORMAL, then neither logical component merge nor logical component recovery is being attempted. LC2 then compares its up list U2 with the up list U1 included in the LUM message. If the set $U2' = U2 \cap U1$ is not empty, then LC2 has not yet sensed that the nodes in U2′ belong to another logical component. In this case, all

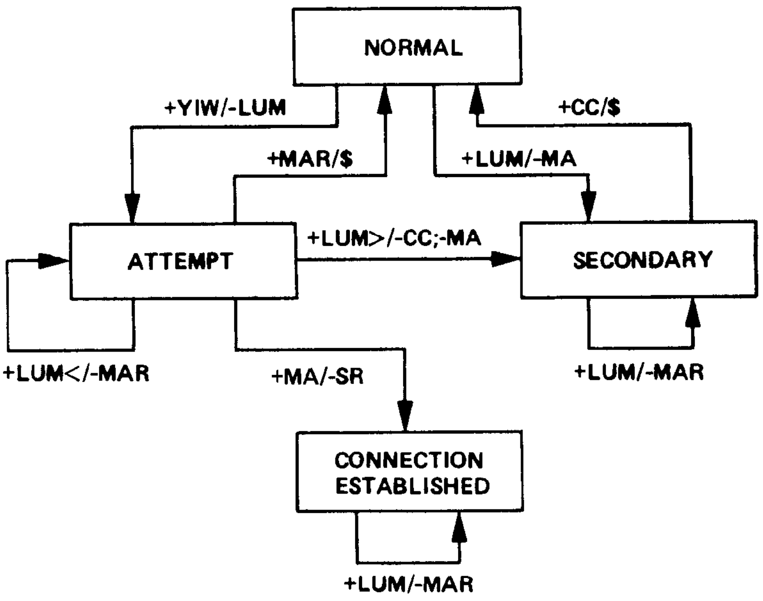


Fig. 3. State transition diagram for P-S connection establishment. A plus (+) sign indicates reception of a message and a minus (−) sign indicates transmission of a message. The sign < indicates that the message in question originates from a lower priority source, while > indicates a higher priority site. The dollar sign indicates that no action is taken due to a state transition.

the locks in the LOCK table of LC2 which are not local anymore with respect to the up list (U2 − U1) should be released and the corresponding transactions should be aborted. After this is done, LC2 sends a "MERGE ACCEPTED" or MA message to LC1 and changes its internal state to SECONDARY. The MA message should contain the up list U2 so that LC1 can compare it to U1 and repeat the same procedure carried out by LC2. This entails releasing all the locks in the LOCK table of LC1 which are not local with respect to the up list (U1 − U2). Upon receipt of the MA message the connection is considered to be successfully established by LC1. If the status of LC2 is not NORMAL, then a "MERGE ATTEMPT REJECTED" or MAR message is sent to LC1 which will either retry later or will try a connection with another LC.

The above interconnection strategy could clearly allow undesirable race conditions to occur, such as having two LCs trying to play the role of primary, leading the system into deadlock situations. To avoid this problem, we assign a site-dependent priority to each LC (no two sites have the same priority). LUM messages from lower priority LCs are rejected. LUM messages from higher priority LCs, if received while the connection has not yet been completed, i.e., the MA message has not been received, cause the connection being attempted to be broken. To this end the primary sends a "CLOSE CONNECTION" or CC message to its intended secondary.

That the protocol outlined above satisfies conditions C1 and C2 is proved in Section 5.1. Figure 3 shows a state transition diagram describing the interconnection protocol. This protocol is the same for every node. Node labels are STATUSes, while arc labels are of the form R/T where R is the message whose arrival triggers the transition and T is a sequence of actions (transmission of messages) which occur as a consequence of the transition.

After a P-S connection has been established between LC1 and LC2, they will not accept any more new lock or lock release requests from nodes in their components and will complete all outstanding ones. An outstanding request is one for which all AL or AR messages have been already sent but not all the corresponding LA or RA messages have been received. After all outstanding requests have been completed by LC2 it sends to LC1 a "READY TO MERGE" or RTM message containing as arguments the up list U2 and the LOCK table at LC2 which now is the same for all nodes in C2. The receipt of the RTM message by LC1 marks the end of the reconnection detection phase.

The merge phase will construct the union of the LOCK tables at both components. Notice that up to this point no permanent change has been made to any LOCK table, nor up list of any node. LC1 sends a "SUBSTITUTE YOUR TABLE" or SYT message for a cycle through the set of nodes in TEMP_U = U1 ∪ U2. The SYT message is the agent which confirms the merge of the two components by taking within it the new LOCK table for the component. Also, the up lists are updated and LC1 becomes the new LC of the new logical component.

4.6 Robustness of LCM

THEOREM 5. *The logical component merge (LCM) algorithm is robust.*

PROOF. The completion point for the LCM algorithm is the point where the SYT message has already been received and accepted by one LLC.

Let LT_i, U_i, and LC_i be, respectively, the LOCK table at site *i*, the up list at site *i*, and the LC identification as known by site *i*. It is worth observing that changes to the values of LT_i, U_i, and LC_i at any site *i* other than the LC1 site are only made upon receipt of the SYT message.

Let us examine the possible cases of crashes before the completion point:

Case 1. Crashes during the reconnection detection phase. A crash of either LC1 or LC2 in this phase will cause LCM to be aborted and an LCR to be started at the component who had an LC crash. Since no LOCK table nor up list has been changed so far, this is a terminal crash. Since LC1 and LC2 are the only processes involved in this phase, we conclude that this phase is robust.

Case 2. Crashes during the merge phase. A crash of LC1 during this phase will interrupt LCM and start LCR for component C1. As no permanent changes have been made already, this is a terminal crash. A crash of any other node (including LC2) clearly does not affect any other node nor the mutual consistency of the merged logical component. □

5. DISJOINTNESS OF THE RECOVERY ALGORITHMS

We show here that there is no interaction between the three recovery algorithms. To that effect one has to show that

(a) LCM is done pairwise.
(b) LCR, LCM, and SNR are mutually exclusive.

To verify condition (a) we only need to show that conditions C1 and C2 stated in Section 4.5 are satisfied by the P-S connection protocol. This verification is done in Section 5.1. Condition (b) is shown to hold in Section 5.2. We shall assume that messages received by a node that are not compatible with the state of that node are ignored by it.

5.1 Disjointness of LCMs

Consider a directed graph G whose vertex-set is the set of LCs and which has two distinct types of arcs, namely e-arcs and a-arcs. There is an e-arc from vertex i to vertex j if there is an established P-S connection between vertices i and j, vertex i being the primary. Equivalently, an e-arc from vertex i to vertex j is said to be created in G whenever vertex i enters the CONNECTION ESTABLISHED state (see Figure 3). There is an a-arc from vertex i to vertex j if vertex i is attempting a P-S connection to vertex j. Such an a-arc is created as soon as vertex i enters the ATTEMPT state (see Figure 3). The graph G displays the pattern of established and attempted connections. Let e-G be the subgraph obtained from G by considering only e-arcs of G and a-G be the one obtained by taking only the a-arcs.

Conditions C1 and C2 can now be rephrased as follows:

C1.1. $0 \leq \text{indegree}(v) \leq 1$ and $0 \leq \text{outdegree}(v) \leq 1$ for all v in e-G.

C2.1. $\text{indegree}(v) * \text{outdegree}(v) = 0$ for all v in e-G.

Every a-arc will either be deleted from G when the attempted connection is broken or will become an e-arc if the connection is successfully established. We want to prove the following:

THEOREM 6. *Given a graph G whose e-graph satisfies conditions C1.1 and C2.1, the new e-graph obtained from G as new connections are established also satisfies those conditions.*

PROOF. It can easily be seen, from the protocol specification, that condition C1.1 is satisfied not only by the initial e-graph but also by the graph G, since

(a) If there is already a connection between vertices i and j or one is being attempted, no new connection is attempted by either vertex i or vertex j.
(b) If a connection has already been established or is being attempted, the secondary will reject all further attempts.

It thus remains for us to examine all the possible cases in which condition C2.1 could conceivably be violated in G and show that the resulting e-graph obtained when one or more a-arcs become e-arcs still satisfies this condition. There are four possible cases, two of which can never happen due to the protocol specification, while the remaining two have to be examined. Given any three vertices a, b, and c, the four possible cases are

(a) (a,b) and (b,c) are e-arcs.
(b) (a,b) is an e-arc and (b,c) is an a-arc.
(c) (a,b) is an a-arc and (b,c) is an e-arc.
(d) (a,b) and (b,c) are a-arcs.

Cases (a) and (b) are the impossible ones. In case (c) the attempted connection between a and b will fail since there is an established connection from b to c (see the self-loop at the CONNECTION ESTABLISHED state of the diagram of Figure 3). Therefore, arc (a,b) will disappear. In case (d) nodes a and b are in the ATTEMPT state. If (a,b) becomes an e-arc we can see that the transition labeled LUM/CC;MA from state ATTEMPT to the state SECONDARY is taken at vertex b, causing the attempted connection (b,c) to be broken. Therefore arc (a,b) becomes an e-arc while arc (b,c) disappears. On the other hand, if (b,c) becomes an e-arc in the first place we are back to case (c) which was already examined. □

We take the opportunity here to prove that the P-S connection protocol is such that all the a-arcs in G will, in a finite time (of the order of magnitude of the transmission delay time in the network) either disappear or become e-arcs. In other words, the P-S connection protocol is deadlock-free.

THEOREM 7. *The P-S connection protocol is deadlock-free.*

PROOF. We must prove that there can be no long-lasting cycles in G. The interesting case is, of course, that of cycles made only out of a-arcs, since as shown in the previous theorem, any a-arc adjacent to an e-arc will disappear in a finite time.

Consider a cycle in a-G and two adjacent a-arcs (a,b) and (b,c) in the cycle. Vertices a and b are in the ATTEMPT state. There are only two possible cases to consider:

Case 1 [PRIORITY(a) > PRIORITY(b)]. In this case, if the "MERGE ACCEPTED" message from vertex c is received by b before the "LET US MERGE" message from a then (b,c) becomes an e-arc and (a,b) disappears.

Case 2 [PRIORITY(a) < PRIORITY(b)]. Here, arc (a,b) will disappear since a has lower priority than b.

In any event, the cycle will eventually be broken. Note also, that vertex c could be the same as a and the above analysis is still valid. □

5.2 Disjointness of LCR, LCM, and SNR

We first define a *node state transition diagram* as a directed graph whose vertices are states of a network node and whose arcs represent transitions between states. The state of a node i is the 3-tuple [STATUS$_i$, LC$_i$, U$_i$], where LC$_i$, U$_i$ are as defined before. STATUS$_i$ is the status of the component to which site i is attached as viewed by site i. NORMAL status indicates that neither LCR nor LCM is in progress; RECOVERY means that LCR is taking place and QUIESCENT indicates that LC$_i$ is rejecting further requests. The labels on the arcs specify the conditions upon which a transition between two states occurs. These conditions can either be a crash detection or a message arrival. The diagram shown in Figure 4 shows all possible state transitions for a node, other than LC1, which is in a component C1, with LC equal to LC1 and up list equal to U1. From every state there is a transition to the DOWN state. These transitions are not represented in the diagram for obvious reasons.

The state [NORMAL, LC$_j$, U$_j$] is a state which resulted from a successful merge of component C1 with another component, for instance C2. The state [NORMAL, LC$_i$, U$_i$] is a state which resulted from a successful logical component recovery.

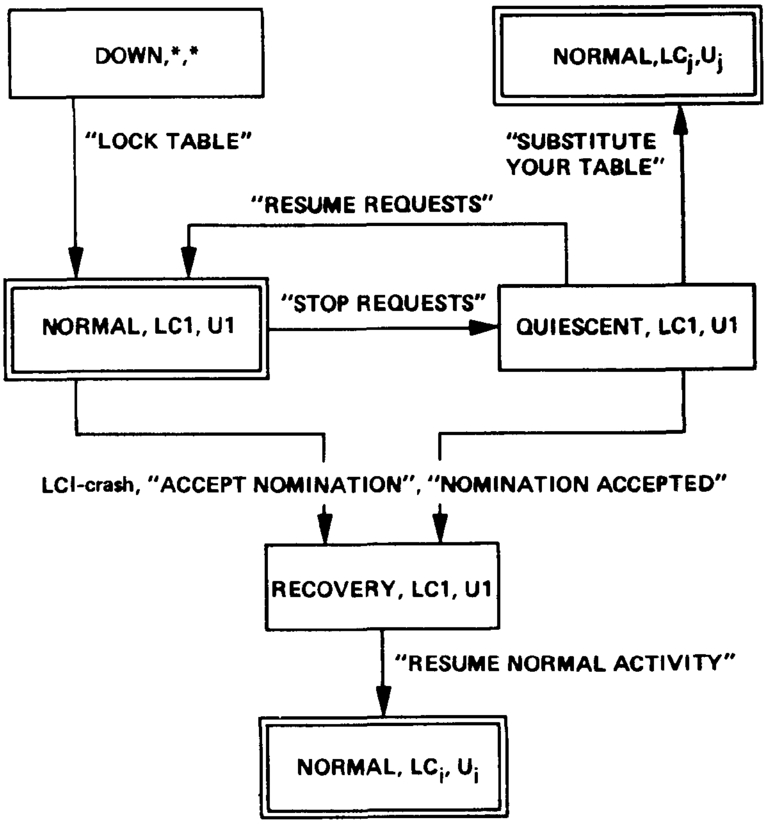


Fig. 4. Node state transition diagram. The following relationships are observed: $U_j = U1 \cup U2$ where LC2 is in U2. $U_i \subseteq U1$. LC_i is in U_i.

By inspection of the diagram we observe that a node can only go from one normal state to a different normal state after one and only one recovery mechanism has been completed. Therefore, there is no interaction among the three recovery mechanisms.

6. LOGICAL COMPONENT MUTUAL CONSISTENCY

Let us show here that the CLC protocol (including the recovery mechanisms) is such that the set of logical components into which the network is partitioned is mutually consistent.

THEOREM 8. *The set of logical components into which the network is partitioned is mutually consistent.*

PROOF. By Theorem 3 each one of the logical components is internally consistent. It remains for us to prove that there can be no lock present at any LOCK table of any component which conflicts with another such lock of any other component. This theorem is trivially true when there is only one logical component. Further net partitioning does not destroy this property since locks are only granted if they are local to a component, which implies that they do not conflict with any other lock granted at any other component. □

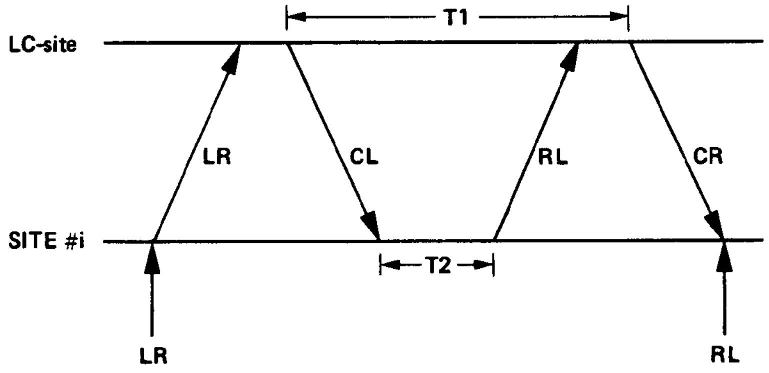


Fig. 5. T1 is the time interval during which a lock is considered to be active at the site where the LC is located. T2 is the time during which the same lock is considered to be active at the requesting site #1.

7. DATABASE CONSISTENCY

We show here that given a deadlock-free consistency-preserving locking mechanism for a *centralized database* (*CDB*), the CLC protocol can be used to implement an equivalent robust, deadlock-free consistency-preserving locking mechanism for a *distributed database* (*DDB*). A database is said to be in a consistent state if all the data items satisfy a set of assertions or consistency constraints. A transaction is a sequence of accesses which take the database from a consistent state into another consistent state. Thus a transaction is the unit of consistency. Let us define an access as the pair (P,a) where P is a logical description of the portion of the database to be accessed and a is an access mode (e.g., read, write, delete, etc). If all the locks are granted by a process which has complete knowledge of every other active lock (as is the case with the LC) and if every access is checked against the LC copy of the LOCK table (this condition will be relaxed later) to see whether the transaction holds the necessary locks, then the "lock scheduler" for a CDB described by Eswaran et al. [6] can be implemented in a straightforward manner with the use of the CLC protocol. Such a locking mechanism has the properties of being robust and preserving the consistency of the DB. Notice that deadlock prevention or detection mechanisms can be carried out by the LC since it has complete control over all activities in its component. Recall that if the network is partitioned into more than one component, locks granted in one of them do not conflict with locks active in others. Therefore, distinct LCs manage disjoint sets of "resources," where a resource here means an individually lockable data item in the DB. So, a deadlock prevention or detection policy can be implemented in each LC independently of all the others.

The requirement that every access be checked against the LOCK table at the LC site can be relaxed in favor of having the access checking done locally. In order for this to be possible a lock must be considered to be active at a given site *i* for a time interval T2 contained in the time interval T1 during which the lock is active at the LC site; otherwise some portions of the DB could be locked in conflicting modes for different transactions. Figure 5 shows a double time axis diagram displaying time at the LC site and at a given site *i* where a lock request is originated. T1 starts when the "CONFIRM LOCK" message is sent to every relevant site in the component and ends with the broadcast of the "CONFIRM RELEASE" message. T2 starts with the arrival of a CL message at site i. Although a lock is only removed from a LOCK table when the corresponding "CONFIRM RELEASE" message arrives, it can be flagged as "waiting for removal" as soon as a "RELEASE LOCK" message is sent from the LC to site i. For access checking purposes, all flagged locks must be considered as nonactive. The extra precaution that must be taken in this case is to unflag all flagged locks after LCR has taken place.

8. PERFORMANCE RESULTS

Some of the results of the cost and delay analysis for the CLC protocol [10] are presented here. The update model used in this analysis is such that some of the previously defined messages are grouped into a single physical message. In order to illustrate this grouping of messages, we will use the abbreviations for messages given in Appendix A. Then, we have the following physical messages:

(1) application program $\rightarrow$ LC: {LR, update request, RL}
(2) LC $\rightarrow$ LLC_i: {AL, AR}
(3) $LLC_i \rightarrow$ LC: {LA, RA}
(4) LC $\rightarrow$ LLC_i: {CL, do update, CR}
(5) LC $\rightarrow$ application program: {update done}

Although some messages are now grouped together into a single physical message, they are still processed as separate messages as if they were received in the order indicated above.

These results indicate that the average update delay, D_{updt}, compares favorably with other schemes [5, 12] since it does not depend directly on the size of the network for many network topologies of interest. The expression for the average update delay is given by

$$D_{updt} = 2\tau + 3\tau_{max} + W$$

where τ is the average message delay introduced by the network between two distinct sites, τ_{max} is the average maximum delay between a sender and just the several destinations involved in the update operation, and W is the average waiting time for a lock request to be granted at the LC.

Lower and upper bounds for the average recovery delay, R, are given by

$$R \geq (n + 1)\tau + 3\tau_{max}$$

and

$$R < [a(n - 2) + n + 1]\tau + 3\tau_{max}$$

where n is the number of sites in the network and a is the ratio τ_{out}/τ where τ_{out} is the time after which a nominator assumes that the nominee is down and sends another "ACCEPT NOMINATION" message to the next site in the nomination order.

The average communications cost, C_{updt}, incurred by an update is

$$C_{updt} = (3k - 1)M.$$

If the LC is one of the sites involved in the update request and it is

$$C_{\text{updt}} = (3k + 2)M \quad \text{otherwise},$$

where k is the number of sites involved in the operation (and not the total number of sites in the network) and M is the average communications cost per message. Lower and upper bounds for the recovery cost C_{rec} are given by

$$C_{\text{rec}} \geq (5n - 4)M$$

and

$$C_{\text{rec}} < (6n - 6)M.$$

9. EXTENSION

It has been observed in most of the existing distributed systems that a large percentage of the generated transactions is *local*, in the sense that the resources needed to satisfy a given transaction are either located at the site of origin of the transaction or in neighboring sites. This observation suggests that significant savings in terms of communications cost and delay can be achieved if one optimizes the operation of the algorithm to adapt to such a highly skewed distribution of activity. To illustrate the point, consider a set of interconnected computer networks. We believe that in such a case, most of the operations will be confined to one computer network while relatively few operations will cross network boundaries.

This section outlines an extension to the CLC protocol that permits the forms of performance optimization needed for the cases discussed above. The extension, which we call an HCLC (for hierarchical CLC) protocol, consists of a hierarchical organization of resource controllers. A *tree* of controllers is provided where the root is considered to be at level 0 and all the children of a controller at level i are at level $i + 1$ in the hierarchy.

Each controller (except for the leaves) serves as an LC for its children. Also, each controller (except for the root of the hierarchy) acts as an LLC for its parent. Therefore, each controller has to maintain two distinct LOCK tables, which we call parent-LT and child-LT. The parent-LT for the root controller contains one lock for the whole DB in exclusive mode. The child-LT for a leaf is empty.

An intuitive description of the normal operation of the HCLC protocol can be easily understood in the light of an example. Figure 6 shows a three-level hierarchy. Application programs interact with lock controllers K1 and K2 at one level above the leaves (since the leaves are LLCs). This interaction is the same as the AP-LC interaction in the CLC protocol. Actually, application programs are not aware of the fact that the controllers are hierarchically organized. Let a lock request, x, from AP1 be submitted to K1. If x conflicts with any other lock in child-LT(K1) then the lock request is treated in the same way as in the CLC protocol. If there is no conflict, K1's parent-LT is searched for a lock y which covers x. A lock $x1$ is said to *cover* a lock $x2$ if the portion of the DB specified by $x2$ is contained in the portion of the DB addressed by $x1$ and if the lock mode specified by $x1$ is not weaker than the lock mode in $x2$. The existence of a lock such as y in parent-LT(K1) indicates that K1 currently has control over the resources requested by AP1. If y is found, the lock request x can be granted and

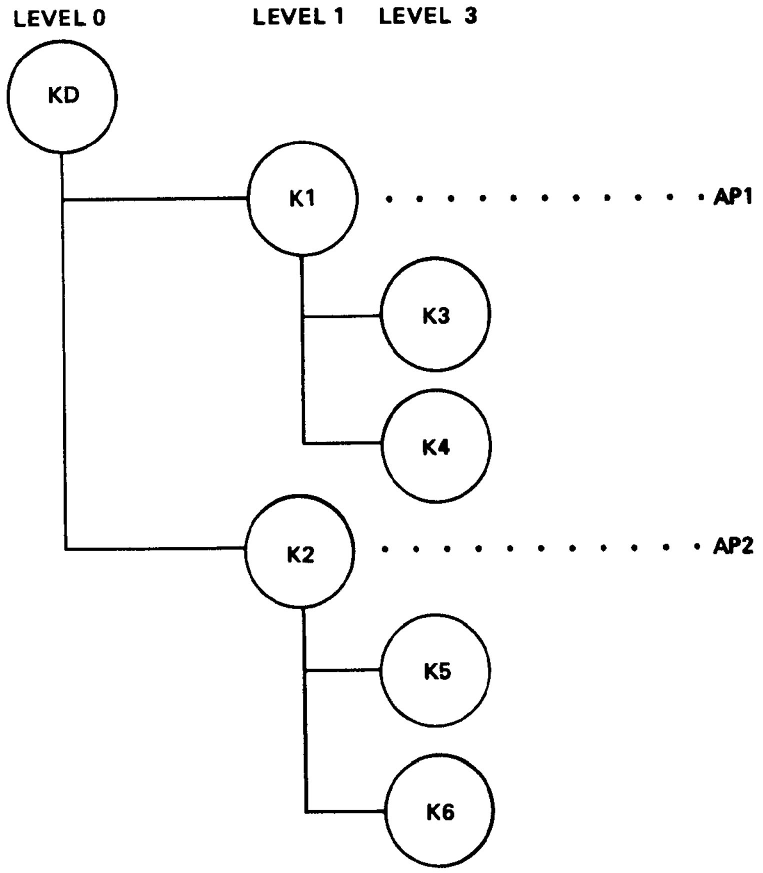


Fig. 6. Hierarchy of lock controllers. AP1 and AP2 are application programs.

to this end K1 interacts with K3 and K4 in the same way as an LC interacts with the LLCs in its component. On the other hand, if y cannot be found, the lock request x is submitted by K1 to K0. K0 will act with respect to K1 and K2 in the same way that K1 did with respect to K3 and K4. The difference in this case is that since K0 is the root there is a lock in parent-LT(K0) for the whole DB in exclusive mode. This lock covers any other lock.

In an HCLC protocol, locks may be released either *explicitly* or *automatically*. Locks in child-LT(K_i), for $i = 1, 2$, are released explicitly upon request from APs using the same mechanism described in the CLC protocol. Locks in parent-LT(K_i), for $i = 1, 2$, can be released automatically as soon as there are no locks in the corresponding child-LTs which depend upon them. To this end, each lock y, in parent-LT(K), for any controller K, has associated with it a list of locks in child-LT(K) covered by y. Also, each lock x in a child-LT(K) points to the lock y in parent-LT(K) which covers x. When a lock x is explicitly released from child-LT(K1) the lock list for its corresponding lock, y, in parent-LT(K1) is appropriately updated. Whenever this list becomes empty, a release request may be automatically generated by K1 and submitted to K0. In general, the automatic release of locks can be propagated up to the root.

This hierarchical protocol can be easily adjusted by policy decisions both to delay such releases, and to establish early locks at higher levels in anticipation of local lock requests. Lock management analogous to LRU-like memory management policies are obvious policy candidates.

For the set of interconnected networks, a three-level hierarchy could be constructed as follows. There is one LC per computer network, all of them at level 1. Their children, at level 2, are their corresponding LLCs. Finally, the root is any site acting as a global controller for the entire collection of computer networks.

An interesting property of the proposed extension is that there is always one controller which is able to detect the existence of a cycle in the lock-request graph. This controller is the common ancestor, with the largest level number, to all the controllers where requests in the cycle were originated. In the example of Figure 6, the common ancestor to K1 and K2 is K0.

Crash recovery algorithms for the HCLC protocol must include mechanisms to reconstruct the hierarchy, in addition to the recovery mechanisms present in the CLC protocol.

10. CONCLUSIONS

This paper outlines what we believe to be a fairly general solution to synchronization issues in distributed systems in the face of asynchronous unplanned failures. The algorithms and protocols for normal operation and recovery are robust with respect to the criteria set up at the beginning of this paper. We are unaware of any other synchronization protocols which simultaneously satisfy each of those requirements.

The work is primarily suitable for environments in which the cost, including delay, of sending messages is not high relative to the operations which are to be performed once locking is complete. Locally distributed systems often provide examples of such an environment. Geographically distributed networks also fall into this category if the amount of work to be performed after locking is significant relative to the communications cost.

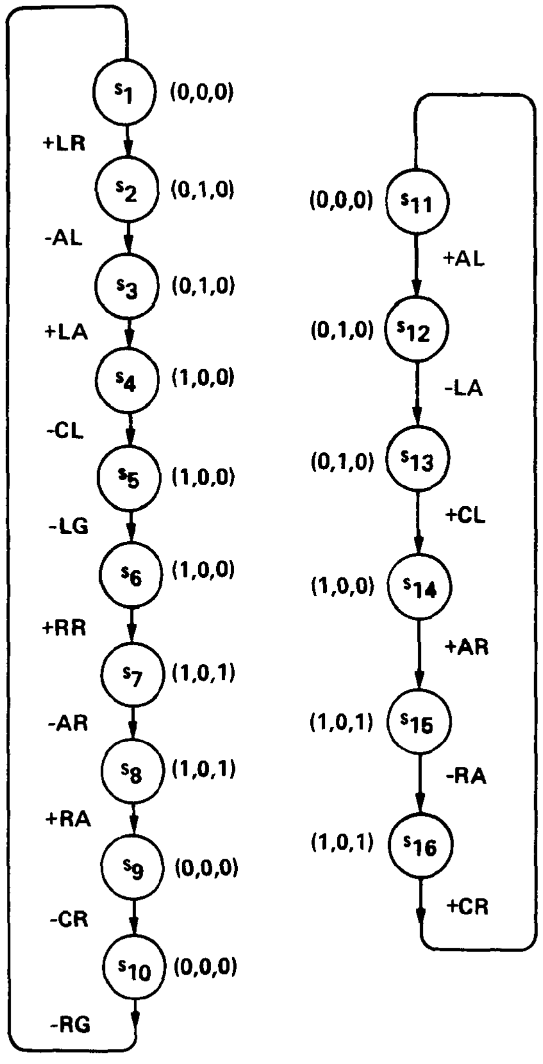
The protocols are also best suited for usage behavior that cannot be directly characterized in advance. It is assumed that query and update activity will be largely ad hoc in nature—the more general case which has been receiving increasing attention in recent years.

The presentation of any substantial protocol would not be complete without an outline of a proof that the protocol is correct with respect to its desired properties. A significant portion of this document is therefore devoted to that purpose.

In conclusion, these protocols should help demonstrate the practicality of integrated cooperation of activities in distributed systems.

APPENDIX A. STATE TRANSITION DIAGRAMS

The operation of the lock granting and lock releasing algorithms can be described by a pair of interacting graphs or state transition diagrams as shown in Figure 7. This technique was introduced by Zafiropulo [13] as a tool for modeling computer communications network protocols. There is one state transition diagram (STD)



(a) (b)

Fig. 7. (a) STD for the LC. (b) STD for an LLC.

for the LC (Figure 7(a)) and one for a local lock controller LLC$_i$ (Figure 7(b)). State transitions are triggered by events such as message transmission of reception. Each STD has a quiescent or *initial state*. As already mentioned in Section 2, the LC does not grant conflicting locks, so that we can consider the operation of the protocol as if there were only one lock request. Let x be such a lock. It is useful to associate with each state s in the STD a 3-tuple $\langle a,b,c \rangle$ where a, b, and c are binary variables which indicate whether the lock x is in the LOCK table, in the L-list, or the R-list, respectively, when the protocol is in state s. For instance, the tuple (1,0,1) indicates that the lock x is in the LOCK table and in the R-list. Labels on the state transition arcs represent conditions or events upon which the transition takes place. These conditions indicate either message transmission or reception. The following abbreviations for message names were used in Figure 7:

From AP to LC

LR: LOCK REQUEST
RR: RELEASE REQUEST

From LC to AP

LG: LOCK GRANTED
RG: RELEASE GRANTED

Among LC and LLCs

AL: ACCEPT LOCK
LA: LOCK ACCEPTED
CL: CONFIRM LOCK
AR: ACCEPT RELEASE
RA: RELEASE ACCEPTED
CR: CONFIRM RELEASE

There are state transitions in one STD which have a companion in the other STD, i.e., transmission of a message in one of them and reception of the same message in the other. We label the transitions with "signed" message names. A positive sign represents a message reception and a negative one indicates a message transmission.

Let a *transition cycle* be a path beginning and ending in the initial state. Let a *conversation* $[A,B]$ be a pair of transition cycles A and B such that each of them belong to different STDs. Let us define an *event sequence* of a transition cycle A as the sequence of labels of the transitions in A which correspond to internal[5] events only.

A conversation $[A,B]$ is said to be *synchronized* if the event sequence of A is equal to the event sequence of B except for the signs in the events which are reversed.

From the description of the protocol one can easily draw the STDs in Figure 7. Actually an STD may be considered as a protocol specification. Let us define a global state S for a set of k STDS, STD_1 through STD_k, as a k-tuple $(s_1, s_2, \ldots, s_k)$ where s_i, for $i \in \{1, \ldots, k\}$, is the 3-tuple associated with state s_i in STD_i. A global state is said to be *feasible* if the individual component states for each STD can coexist.

If all the conversations of a protocol are synchronized then the task of finding the set of all global feasible states is fairly straightforward. Consider the synchronized conversation $[A,B]$ shown in Figure 8.

Let $-M$ be a condition on the transition cycle A which triggers the transition from state $s_{a,i}$ into state $s_{a,i+1}$. Also, let $+M$ be the companion condition in the transition cycle B which triggers the transition from state $s_{b,j}$ into state $s_{b,j+1}$. R1

[5] An event is internal if it does not represent an external action such as an exchange of messages with an application program.

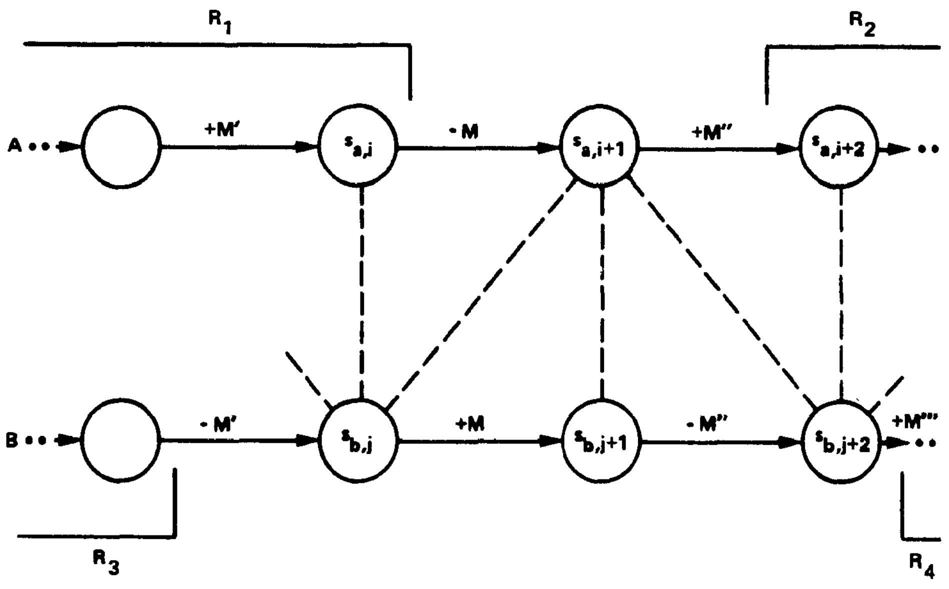


Fig. 8. Synchronized conversation $[A, B]$. The dashed lines indicate global feasible states.

is the set of states in A which precede[6] $s_{a,i+1}$. R2, R3, and R4 have a similar interpretation (see Figure 8). Let us consider first a state which is reached by a transition labeled by a message transmission, say $s_{a,i+1}$, for instance. The set of possible global states which include $s_{a,i+1}$ are

(1) $\{(s_{a,i+1}, x) \mid x \in \text{R3}\}$: these states are not feasible because message M' was not yet sent but already received.
(2) $\{(s_{a,i+1}, x) \mid x \in \text{R4}\}$: these states are not feasible because M''' was not yet sent but already received.
(3) States $(s_{a,i+1}, s_{b,j})$, $(s_{a,i+1}, s_{b,j+1})$, and $(s_{a,i+1}, s_{2k})$ are feasible.

Let us now consider a state which is reached by a transition labeled by a message reception, $s_{b,j+1}$, for instance. The set of all the global states which include $s_{b,j+1}$ are

(1) $\{(s_{b,j+1}, x) \mid x \in \text{R1}\}$: these states are not feasible because message M was not yet sent but already received.
(2) $\{(s_{b,j+1}, x) \mid x \in \text{R2}\}$: these states are not feasible because message M'' was not yet sent but already received.
(3) State $(s_{b,j+1}, s_{a,i+1})$ is feasible.

This example illustrates the rule for generating the set of all the global feasible states in a protocol where all the conversations are synchronized. Namely, for every nonexternal transition M, add to the set of global feasible states the states $(s_{a,i}, s_{b,j})$, $(s_{a,i+1}, s_{b,j})$, and $(s_{a,i+1}, s_{b,j+1})$. If any of the STDs contains transitions which are triggered by external events (e.g., an external request from an appli-

[6] A state s_x is said to precede state s_y if there is a transition from s_x into s_y and if s_y is not the initial state.

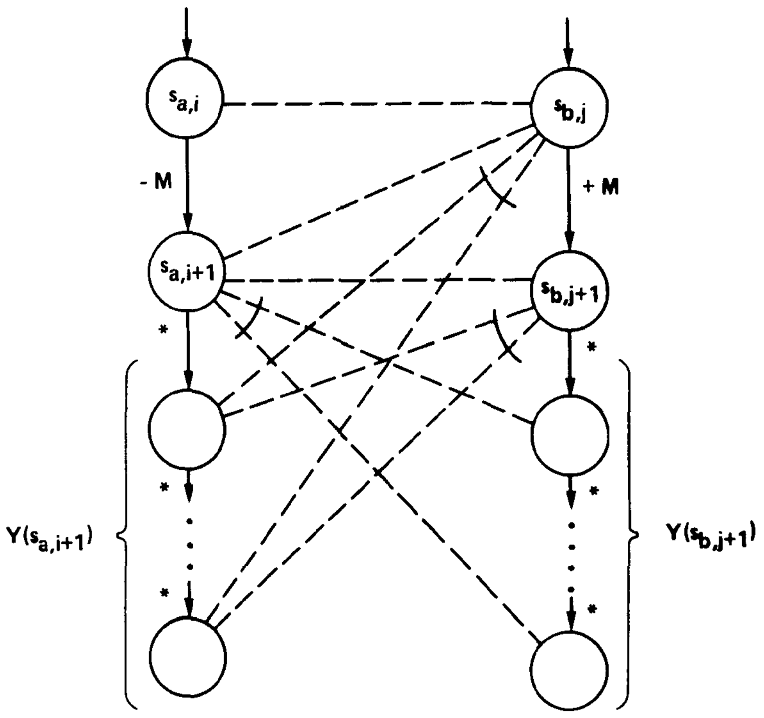


Fig. 9. Graphical representation of the set of rules to find the set of global feasible states associated with transition M. The dashed lines represent external events.

cation program), then the following rules must be added. Let $Y(s)$ be the set of all the states which can be reached from state s through a path containing transitions associated with external events only. For every nonexternal transition M, add to the set of global feasible states the states $(s_{a,i}, s_{b,j})$, $(s_{a,i+1}, s_{b,j})$, $(s_{a,i+1}, s_{b,j+1})$ and the sets

$$\{(s_{b,j}, q) \mid q \in Y(s_{a,i+1})\},$$

$$\{(s_{b,j+1}, q) \mid q \in Y(s_{a,i+1})\}$$

and

$$\{(s_{a,i+1}, q) \mid q \in Y(s_{b,j+1})\}.$$

Figure 9 illustrates these rules.

Using the rules derived above, one obtains the set of feasible states for the grant and release lock protocols in a system with an LC and one LLC. These states are shown in Table I.

In general, a global state for a system with an LC and k LLCs is of the form of $S = (X, x_1, x_2, \ldots, x_k)$ where X is an LC-state and x_i, for $i = 1, \ldots, k$, is an LLC state compatible with the LC state X, according to Table I. For instance, if $X = s_7 = (1, 0, 1)$ the three following are examples of global states when we have three LLCs: $(s_7; s_{13}, s_{14}, s_{13})$, $(s_7; s_{13}, s_{13}, s_{14})$, and $(s_7; s_{13}, s_{13}, s_{13})$.

Table I

Feasible global states	(LC state; LLC$_i$ state)
(s_2, s_{11})	(0, 1, 0; 0, 0, 0)
(s_3, s_{12})	(0, 1, 0; 0, 1, 0)
(s_2, s_{16})	(0, 1, 0; 1, 0, 1)
(s_4, s_{13})	(1, 0, 0; 0, 1, 0)
(s_5, s_{14})	(1, 0, 0; 1, 0, 0)
(s_7, s_{13})	(1, 0, 1; 0, 1, 0)
(s_7, s_{14})	(1, 0, 1; 1, 0, 0)
(s_8, s_{15})	(1, 0, 1; 1, 0, 1)
(s_9, s_{16})	(0, 0, 0; 1, 0, 1)
(s_{10}, s_{11})	(0, 0, 0; 0, 0, 0)

APPENDIX B. PROOF FOR ASSERTIONS

Proof for Assertion 1

Let k be the number of relevant LLCs in the component and let $S = (X, x_1, x_2, \ldots, x_k)$ be a feasible global state. The validity of this assertion can be readily verified by examination of Table I. If a lock is in the LOCK table of the LC the state X is equal to (1,0,0) if there are no pending lock release requests associated with the lock in question. But for $X = (1,0,0)$, the only possible LLC states are (0,1,0) and (1,0,0). Therefore, the lock is present at every site and the assertion is proved.

Proof for Assertion 2

Analogously to the proof of Assertion 1, this assertion is easily proved by examination of Table I. The only possible LLC states associated with the LC state $X = (0,0,0)$ are (1,0,1) and (0,0,0). Therefore, the release request in question is present at every site and the assertion is proved.

Proof for Assertion 3

We need to show that at the end of the notification phase, every site will end up with the same value for the trial list T and consequently the same value for new_LC. Before doing so, we must prove the following two assertions.

Assertion 3.1. Given two trial sequences, one of them must be a prefix of the other. In other words, let $X = x_1, x_2, \ldots, x_p$ and $Y = y_1, y_2, \ldots, y_p, y_{p+1}, \ldots, y_{p+q}$ then $x_i = y_i$ for $i = 1, \ldots, p$.

PROOF. This assertion follows directly from the fact that the nomination is done following the nomination order and starting at the node that follows the crashed LC in this order.

Assertion 3.2. The sequence, $T_1, T_2, \ldots, T_m$ of trial sequence values taken by the trial list T is such that if $T_i = i_1, i_2, \ldots, i_p$, then $T_{i+1} = i_1, i_2, \ldots, i_p, i_{p+1}, \ldots,$

i_{p+q} for $i = 1, \ldots, (m - 1)$. In other words, T_i and T_{i+1} have the same prefix, namely, $i_1, \ldots, i_p$.

PROOF. By Assertion 3.1, either T_i is a prefix of T_{i+1} or vice versa. If T_i is empty this assertion is trivially true. So, from this point on assume that T_i is not empty. Assume now that this assertion is not valid. Therefore, it must be the case that if $T_{i+1} = j_1, \ldots, j_p$ then $T_i = j_1, \ldots, j_p, j_{p+1}, \ldots, j_{p+q}$ for $q > 0$. But, as T_i is not empty, in order for T to be set to the value of T_{i+1}, it must be the case that, during execution of the algorithm in Section 4.1, j_{p+1} was not in T_i. This contradicts our assumption and thus proves Assertion 3.2.

We can now prove Assertion 3. Assume that this assertion is not true. So, there are at least two sites i and j for which its T values are different at the end of the notification phase. Let $T_i = X, i_1, \ldots, i_p$, and $T_j = X$ where X is a common prefix. If T_i has the value $X, i_1, \ldots, i_p$ and the notification phase has already ended, then an NA message with trial sequence equal to T_i must have passed through node j. By that time, the algorithm in Section 4.1 would have changed the value of T_j to that of T_i. This contradicts our assumption and shows that all T and consequently new_LC values will end up being the same at the end of the notification phase. That the value for new_LC is the identification number of the latest LC to be nominated follows directly from Assertion 3.2.

Proof for Assertion 4

In order to prove this assertion we must consider all the possible global states in which the set of LLCs can be left at when a crash occurs. Table I gives us the set of possible LLC states for each LC state. Therefore, when a crash occurs, the resulting global state is one of the possible combinations of LLC states associated with the state of the crashed LC just before the crash.

The list of all the possible combinations was derived from Table I and the 11 resulting cases are listed below. Each case is represented by a set Q of LLC states where the set of global states associated with Q is such that there is at least one LLC state associated with each element of Q.

Case 1: (0,0,0).
Case 2: (1,0,0).
Case 3: (0,1,0).
Case 4: (1,0,1).
Case 5: (0,0,0; 0,1,0).
Case 6: (0,1,0; 1,0,0).
Case 7: (0,0,0; 1,0,1).
Case 8: (1,0,0; 1,0,1).
Case 9: (0,1,0; 1,0,1).
Case 10: (0,0,0; 0,1,0; 1,0,1).
Case 11. (0,1,0; 1,0,0; 1,0,1).

The actions taken in each of the above cases by the algorithm that builds the sets L and R are summarized in Table II.

Assertion 4 states that a lock p is either in all the LOCK tables of S(p) or it is in none of them. Let us prove this statement as a lemma.

Table II. Actions Taken by the Algorithm that Builds the Sets L and R

Cases	Actions
1 and 2	No action
3, 5, and 6	Add lock to the set L
4, 7, and 8	Add lock to the set R
9, 10, and 11	If seq # (0, 1, 0) > seq # (1, 0, 1) Then add lock to the set L Else add lock to the set R

LEMMA. *At the end of the LCR mechanism the following is true. If the lock p is in one LOCK table of a site in S(p) then it must be in the LOCK table of all of them.*

PROOF. Assume that it is not in at least one LOCK table but that it is in some of them. Then, this lock was not included in the set L otherwise it would have been added to all the LOCK tables. In a similar vein, it was not included in the set R otherwise it would have been deleted from all the LOCK tables. Therefore, the lock was in no L-list nor R-list when the LCR mechanism started. This eliminates Cases 3 through 11. We are then left with Cases 1 and 2. Case 1 is also eliminated because by assumption the lock is in at least one LOCK table. Case 2 is also eliminated because by assumption the lock is not in at least one LOCK table. All the cases have been eliminated. This is a contradiction and proves the lemma.

The complete statement of Assertion 4 now follows directly frm the algorithm which builds the sets L and R. As indicated in Table II, the actions taken in each case are such that maximum forward progress is achieved in each case. In other words, locks that would have been granted are added to all the LOCK tables and locks that would have been released are deleted from all the LOCK tables. Thus, Assertion 4 is proved.

REFERENCES

1. ALSBERG, P.A., BELFORD, G., DAY, J.D., AND GRAPA, E. Multy copy resiliency techniques. CAC Doc. 202, Ctr. for Advanced Computation, Univ. Illinois at Urbana-Champaign, May 1976.
2. BERNSTEIN, P.A., SHIPMAN, D.W., ROTHNIE, J.B., AND GOODMAN, N. The concurrency control mechanism of SDD-1: A system for distributed databases (the general case). Tech. Rep. CCA-77-09, Computer Corporation of America, Cambridge, Mass., Dec. 1977.
3. BUNCH, S.R. Automated backup. Prelim. Res. Study Rep., CAC Doc. 162 (JTSA Doc. 5509), Ctr. for Advanced Computation, Univ. Illinois at Urbana-Champaign, May 1975.
4. CHU, W.W. Performance of file directory systems for databases in star and distributed networks. *Proc. Nat. Comptr. Conf.*, 1976, pp. 577–587.
5. ELLIS, C.A. Consistency and correctness of duplicate database systems. ACM/SIGOPS Operating Systems Review, Vol. 11, 5, *Proc. 6th Symp. Operating Systems Principles*, Nov. 1977.
6. ESWARAN, K.P., GRAY, J.N., LORIE, R.A., AND TRAIGER, I.L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.

7. GRAPA, E. Characterization of a distributed database system. Ph.D. Dissertation, Rep. UIUCDCS-R-76-831, Dep. Comptr. Sci., Univ. Illinois, Urbana, Oct. 1976.
8. GRAY, J.N. Operating systems: An advanced course, Chapter 3.F. In *Notes on Database Operating Systems*. Springer-Verlag, Berlin, 1978, pp. 394–481.
9. LAMPSON, B., AND STURGIS, H. Crash recovery in a distributed data storage system. Tech. Rep., Xerox Palo Alto Res. Ctr., Palo Alto, Calif., 1976. (To be published in *Commun. ACM.*)
10. MENASCE, D.A., POPEK, G.J., AND MUNTZ, R.R. A locking protocol for resource coordination in distributed systems. Tech. Rep. UCLA-ENG-7808, SDPS-77-001 (DSS MDA 903-77-C-0211), Comptr. Sci. Dep., Univ. California, Los Angeles, Oct. 1977.
11. STONEBRAKER, M., AND NEUHOLD, E. A distributed data version of INGRES. Memo ERL M612, Electronics Res. Lab., Univ. California, Berkeley, Sept. 1976.
12. THOMAS, R.H. A solution to the update problem for multiple copy data bases which uses distributed control. Tech. Rep. 3340, Bolt Beranek and Newman, July 1976.
13. ZAFIROPULO, P. Protocol validation by duologue-matrix analysis. Tech. Rep. RZ 816 (#27758), IBM Zurich Res. Lab., Data Communications Ctr., Zurich, Switzerland, Nov. 1977.

Received April 1978; revised November 1979; accepted November 1979