NONBLOCKING COMMIT PROTOCOLS*

Dale Skeen

Computer Science Division
EECS Department
University of California
Berkeley, California

"From a certain point onward there is no longer any turning back. That is the point that must be reached."

- Kafka

ABSTRACT

Protocols that allow operational sites to continue transaction processing even though site failures have occurred are called nonblocking. Many applications require nonblocking protocols. This paper investigates the properties of nonblocking protocols. Necessary and sufficient conditions for a protocol to be nonblocking are presented and from these conditions a method for designing them is derived. Both a central site nonblocking protocol and a decentralized nonblocking protocol are presented.

1. Introduction

Recently, considerable research interest has been focused on distributed data base systems [LORI77, ROTH77, SCHA78, SVOB79]. Several systems have been proposed and are in various stages of implementation, including SDD-1 [HAMM79], SYSTEM-R [LIND79], and Ingres [STON79]. It is widely recognized that distributed crash recovery is vital to the usefulness of these systems. However, resilient protocols are hard to design and they are expensive. Crash recovery algorithms are based on the notion that certain basic operations on the data are logically indivisible. These operations are called transactions.

*This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 78-3596, the U.S. Army Research Office Grant DAAG29-76-G-0245, and the Naval Electronics Systems Command Contract N00039-78-G-0013.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1981 ACM 0-89791-040-0 /80/0400/0133 $00.75

Transaction Management

By definition, a transaction on a distributed data base system is an atomic operation: either it executes to completion or it appears never to have executed at all. However, a transaction is rarely a physically atomic operation, rather, during execution it must be decomposed into a sequence of physical operations. This discrepancy between logical atomicity (as seen by the application) and physical atomicity poses a significant problem in the implementation of distributed systems. This problem is amplified when transaction atomicity must be preserved across multiple failures. Nonetheless, most applications require that a notion of transaction atomicity (above the level of physical atomicity) be supported and made resilient to failures.

Preserving transaction atomicity in the single site case is a well understood problem [LIND79, GRAY79]. The processing of a single transaction is viewed as follows. At some time during its execution, a commit point is reached where the site decides to commit or to abort the transaction. A commit is an unconditional guarantee to execute the transaction to completion, even in the event of multiple failures. Similarly, an abort is an unconditional guarantee to "back out" the transaction so that none of its results persist. If a failure occurs before the commit point is reached, then immediately upon recovering the site will abort the transaction. Commit and abort are

irreversible. See [LIND79] for a discussion on implementing this abstraction of transaction management.

The problem of guaranteeing transaction atomicity is compounded when more than one site is involved. Given that each site has a local recovery strategy that provides atomicity at the local level, the problem becomes one of insuring that the sites either unanimously abort or unanimously commit. A mixed decision results in an inconsistent data base.

Protocols for preserving transaction atomicity are called commit protocols. Several commit protocols have been proposed [ALSB76, HAMM79, LAMP76, LIND79, STON79] The simplest commit protocol that allows unilateral abort is the two phase commit protocol illustrated in figure 1 [GRAY79, LAMP76]. This protocol uses a designated site (site 1 in the figure) to coordinate the execution of the transaction at the other sites. In the first phase of the protocol the coordinator distributes the transaction to all sites, and then each site individually votes on whether to commit (yes) or abort (no) it. In the second phase, the coordinator collects all the votes and informs each site of the outcome. In the absence of failures, this protocol preserves atomicity.

Nonblocking Commit Protocols

Consider what happens in the two phase protocol if both the coordinator and the second site crash after the third site has voted on the transaction, but before the third has received a commit message. There are several possible execution states of the transaction; two are of interest.

First, either of the failed sites may have aborted the transaction. Secondly, all sites may have decided to commit the protocol. In the latter situation, if the coordinator failed between sending commit messages and if the second site failed after receiving a commit message, then the transaction has been committed at the second site. Since site three has no way of determining the status of the transaction at the second site, it can not safely proceed. Instead, execution of the transaction must be blocked at site three until one of the failed sites has recovered.

The two phase commit protocol is an example of a blocking protocol: operational sites sometimes wait on the recovery of a failed sites. Locks must be held on the database while the transaction is blocked.

A protocol that never requires operational sites to block until a failed site has recovered is called a nonblocking protocol.

Termination and Recovery Protocols

When the occurrence of site failures render the continued execution of the commit protocol impossible, then a termination protocol is invoked. The purpose of a termination protocol is to terminate transaction execution as quickly as possible at the operational sites. The protocol, of course, must guarantee transaction atomicity. Clearly, a termination

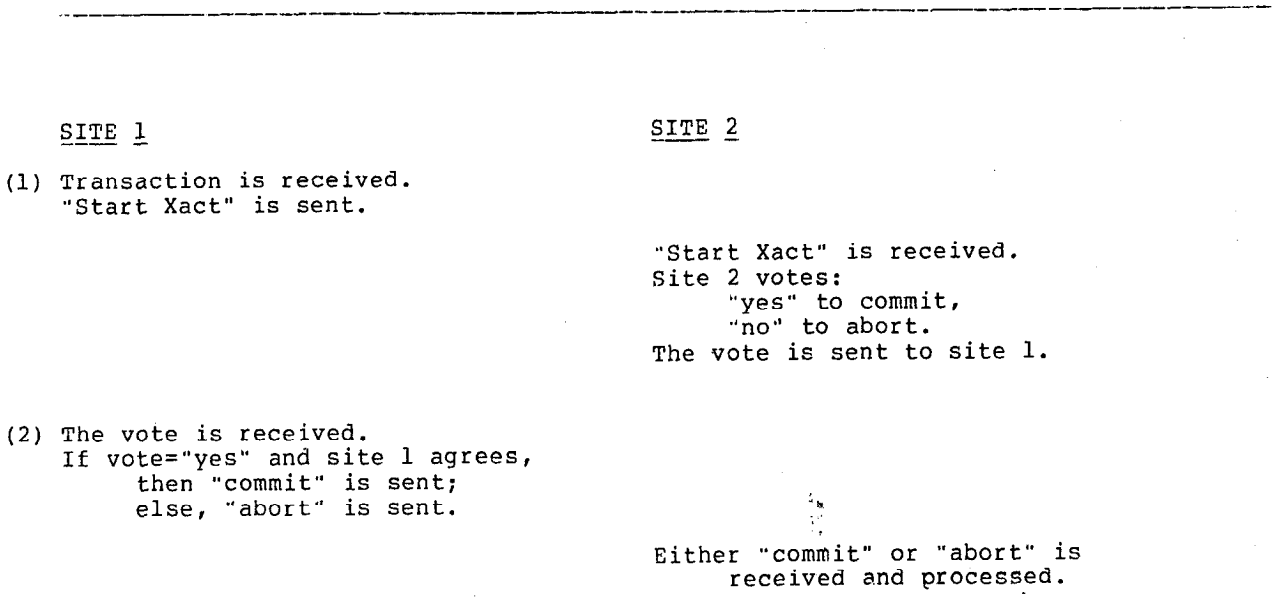

Figure 2. The two-phase commit protocol (2 sites).

protocol can accomplish its task only if a nonblocking commit protocol is used. In section 6, we derive a centralized termination protocol.

The final class of protocols required to handle site failures are called recovery protocols. These protocols are invoked by failed sites to resume transaction processing. Recovery protocols are not discussed in this paper, interested readers are referred to [LIND79, HAMM79, SKEE81a].

In the next section we present the formalisms required in the remainder of the paper. Commit protocols are modelled by finite state automata. The local state and the global state of a transaction are defined.

In the third section two prevalent commit paradigms are presented: the central site model and the decentralized model. It is shown that protocols in both models have synchronization points. This property will be used in designing nonblocking protocols.

In section fourth the major results of the paper are presented. First, necessary and sufficient conditions for a protocol to be nonblocking are derived. Next, we demonstrate that "buffer states" can be added to a protocol to make it nonblocking. For most practical protocols, a single buffer state is sufficient.

In the fifth section we present a protocol invoked to terminate the transaction at the operational sites after the occurrence of (multiple) site failures.

Throughout the paper two assumptions about the underlying communications network are made:

(1) point-to-point communication is possible between two operational sites (i.e. the network never fails),

(2) the network can detect the failure of a site (e.g. by a "timeout") and can reliably report this to an operational site.

2. Formal Model Summarized

In this section, we use a generalization of the formal model introduced in [SKEE81a] to describe commit protocols. Transaction execution at each site is modelled as a finite state automaton (FSA), with the network serving as a common input/output tape to all sites. The states of the FSA for site i are called the local states of site i.

A state transition involves the site reading a (nonempty) string of messages addressed to it, writing a string of messages, and moving to the next local state. The change of local state is an instantaneous event, marking the end of the transition (and all associated activity). In the absence of a site failure, a state transition is an atomic event. State transitions at one site are asynchronous with respect to transitions at other sites.

In figure 2, this model is illustrated for the two phase commit protocol of figure 1. One FSA describes the protocol executed by the coordinator, while the other describes the protocol executed by each slave. Each FSA has four (local) states: an initial state (q_i), a wait state (w_i), an abort state (a_i), and a commit state (c_i). Abort and commit are final states, indicating that the transaction has been either aborted or committed,

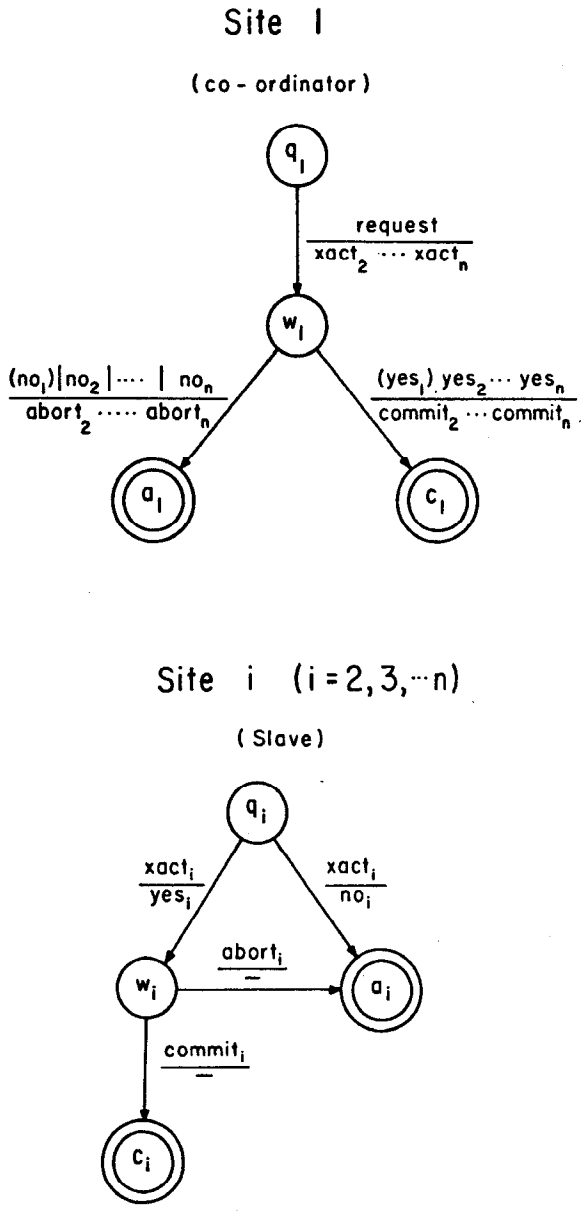

Figure 2. The FSA's for the two phase commit (n sites).

respectively.

Figure 2 also illustrates the conventions used in the remainder of the paper. Local states for site i are subscripted with i. Messages sent or received by a slave are subscripted with that slave's site number.

The finite state automata describing a commit protocol exhibit the following four properties:

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

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

(3) Once a site has made a transition to an abort state, then transitions to nonabort states are not allowed. A similar constraint holds for commit states. Consequently, the act of committing or aborting is irreversible.

(4) The state diagram describing a FSA is acyclic. This guarantees that the protocol executing at every site will eventually terminate.

Protocols are often characterized by the number of phases required to commit the transaction. Intuitively, a phase occurs when all sites executing the protocol make a state transition. The number of phases in a protocol is a rough measure of its complexity and cost (in messages). Distributed protocols generally require at least two phases.

Global Transaction State

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

(1) a global state vector containing the local states of the participating FSA's and

(2) the outstanding messages in the network.

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

The graph of all global states reachable from the initial global state is instrumental in specifying and analyzing protocols. For example, a global state is said to be inconsistent if it contains both a local commit state and a local abort state. Protocols which maintain transaction atomicity can have no inconsistent global states. Figure 3 gives the reachable state graph for the two phase protocol discussed earlier.

A global state is said to be a final state if all local states contained in the state vector are final states. A global

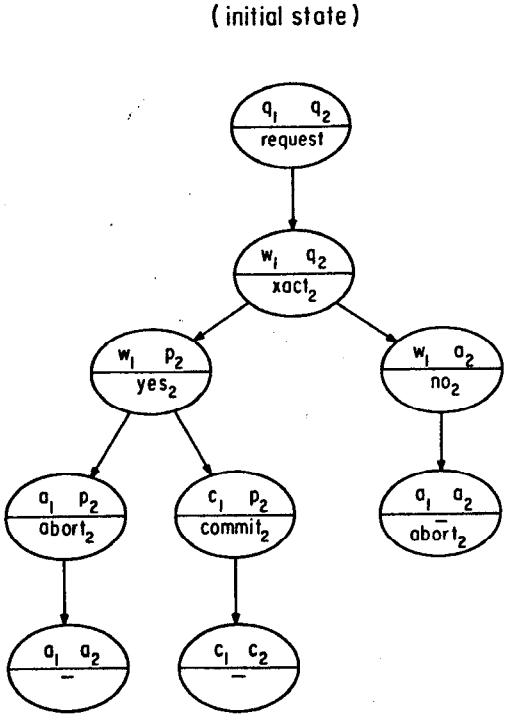

Figure 3. Reachable state graph for the two phase commit protocol.

state is a terminal state if from it there are no immediately reachable successors. A terminal state that is not a final state is a deadlocked state: the transaction will never be successfully completed.

Given that the state of site i is known to be s_i, then it is possible to derive from the global state graph the local states that may be concurrently occupied by other sites. This set of states is called the concurrency set for state s_i[1].

Although the reachable global state graph grows exponentially with the number of sites, in practice we seldom need to actually construct the graph. In subsequent sections, we will be able to infer most properties of the graph by examining properties of the local states.

Committable States

A local state is called committable if occupancy of that state by any site

[1]Formally, the concurrency set of state s_i is the set of all local states s_j, where $i \neq j$, such that s_i and s_j are contained in the same (reachable) global state.

implies that all sites have voted yes on committing the transaction. A site that is not committable is called noncommittable[2]. Intuitively, a site in a noncommittable state does not know whether all the other sites have voted to commit.

In the two phase protocol of figure 2, the only committable state is the commit state (c_i); all other states are noncommittable. Recall, that this protocol is a blocking protocol, and it is common for blocking protocols to have only one committable state. We will assert (without proof) that nonblocking protocols always have more than one committable state.

Site Failures

Since the sending of more than one message is not a physically atomic operation, it can not be assumed that local state transitions are atomic under site failures. A site may only partially complete a transition before failing. In particular, only part of the messages that were to be sent during a transition may, in fact, be transmitted.

Failures cause an exponential growth in the number of reachable global states. Fortunately, it will never be necessary to construct the (reachable) global state graph with failures. In the subsequent sections, any reference to global state graphs will be to graphs in the absence of failures.

3. The Two Paradigms for Commit Protocols

Almost every commit protocol can be classified into either one of two generic classes of commit protocols: the central site class or the (completely) decentralized class. These classes represent two very distinct philosophies in commit protocols. In this section, we characterize and give an example of each class. The examples were chosen because they are the simplest and most renowned protocols in these classes. However, neither example is a nonblocking protocol. In the next section we will show how to extend both of them to become nonblocking protocols.

The Central Site Model

This model uses one site, the coordinator to direct transaction processing at all the participating sites, which we will denote as slaves.

[2]To call "noncommittable" states "abortable" would be misleading, since a transaction that is not in a final commit state at any site can still be aborted. In fact, sometimes transactions in committable (but not commit) states will be aborted because of failures.

The properties of protocols in this class are:

(1) There is a single coordinator, executing the coordinator protocol.

(2) All other participants (slaves) execute the slave protocol.

(3) A slave can communicate only with the coordinator.

(4) During each phase of the protocol the coordinator sends the same message to each slave and waits for a response from each one.

The two phase protocol presented in figures 1 and 2 is the simplest example of a central site protocol. Other examples can be found in [LAMP76, HAMM79, SKEE81a]. Central site protocols are popular in literature because they are relatively cheap, conceptually simple, and robust to most single site failures. Their major weakness is their vulnerability to a coordinator failure.

Property (4) assures that the sites progress through the protocol at approximately the same rate. Let us define this property as follows:

> Definition. A protocol is said to be synchronous within one state transition if one site never leads another site by more than one state transition during the execution of the protocol.

The central site protocol (including both the coordinator protocol and the slave protocol) is "synchronous within one state transition". This property will be used in constructing nonblocking central site commit protocols.

The Decentralized Model

In a fully decentralized approach, each site participates as an equal in the protocol and executes the same protocol. Every site communicates with every other site.

Decentralized protocols are characterized by successive rounds of message interchanges. We are interested in a rather stylized approach to decentralized protocols: during a round of message interchange, each site will send the identical message to every other site. A site then waits until it has received messages from all its cohorts before beginning the next round of message interchange. To simplify the subsequent discussion, during a message interchange we will speak as if sites send messages to themselves.

The simplest decentralized commit protocol is the decentralized two phase commit illustrated in figure 4. All participating sites run this protocol.

(Messages are doubly subscripted: the first subscript refers to the sending site, the second refers to the receiving site.)

In the first phase each site receives the "start xact" message[3], decides whether to unilaterally abort, and sends that decision to each of its cohorts. In the second phase, each site accumulates all the abort decisions and moves to a final state.

Like the central site two phase protocol, the decentralized two phase protocol is synchronous within one state transition. Sites progress through the protocol at approximately the same rate.

4. Nonblocking Commit Protocols

In this section we present the major result of this paper: necessary and sufficient conditions for a protocol to be nonblocking. We then augment the protocols presented in the last section to construct nonblocking protocols.

The Fundamental Nonblocking Theorem

When a site failure occurs, the operational sites must reach a consensus on committing the transaction by examining their local states.

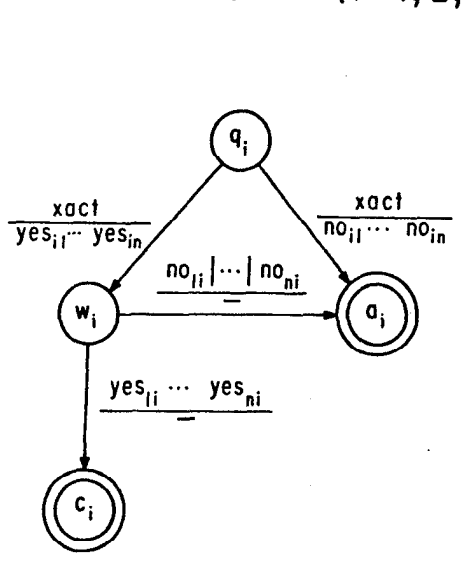

Figure 4. The decentralized two phase commit protocol (n sites).

[3]We do not model the mechanism by which the transaction is distributed to the sites. This is most likely performed by the site receiving the transaction request from the application.

Let us consider the simplest case, where only a single site remains operational. This site must be able to infer the progress of the other sites solely from its local state. Clearly, the site will be able to safely abort the transaction if and only if the concurrency set for its local state does not contain a commit state. On the other hand, for the site to be able to safely commit, its local state must be "committable" and the concurrency set for its state must not contain an abort state.

A blocking situation arises whenever the concurrency set for the local state contains both a commit and an abort state. A blocking situation also arises whenever the site is in a "noncommittable" state and the concurrency set for that state contains a commit state -- the site can not commit because it can not infer that all sites have voted yes on committing, and it can not abort because another site may have committed the transaction before crashing. Notice that both two phase commit protocols can block for either reason.

These observations imply the following simple but powerful result.

> Theorem 1 (the fundamental nonblocking theorem). A protocol is nonblocking if and only if it satisfies both of the following conditions (for every participating site):
>
> (1) there exists no local state such that its concurrency set contains both an abort and a commit state,
>
> (2) there exist no noncommittable state whose concurrency set contains a commit state.

Again, the single operational site case demonstrated the necessity of the conditions stated in the theorem. To prove sufficiency, we must shown that it is always possible to terminate the protocol, in a consistent state, at all operational sites In section 5 we present a termination protocol that will successfully terminate the transaction executed by any commit protocol obeying both conditions of the fundamental nonblocking theorem.

A useful implication of this theorem is the following corollary.

> Corollary. A commit protocol is nonblocking with respect to k-1 site failures (2 < k <= the number of participating sites) if and only if there is a subset of k sites that obeys both conditions of the fundamental nonblocking theorem.

It is obvious that a protocol with k sites obeying the fundamental theorem will be

nonblocking as long as one of those k sites remains operational. (The case where k=2 is a special case that has been examined in [SKEE81b].)

The fundamental nonblocking theorem provides a way to check whether a protocol is nonblocking; however, it does not provide a methodology for constructing nonblocking protocols. In the next section we develop a set of design rules that yield nonblocking protocols. These rules take the form of structural constraints.

Buffer States

The two phase central site (slave) protocol and the two phase decentralized protocol are very similar: they are structurally equivalent, and they are both synchronous within one state transition. These similarities, especially the latter, suggests that a common solution to the blocking problem may exist. Their common structure, which is illustrated in figure 5 for reference, constitutes the canonical two phase commit protocol.

Consider a protocol that is synchronous within one state transition. The concurrency set for a given state in the protocol can contain only the states that are adjacent to the given state and the given state, because the states of the participating sites never differ by more than a single state transition. In the canonical two phase commit protocol, the concurrency set of state q contains q, w and a. The concurrency set for state w contains all of the local states of the protocol.

This observation together with the fundamental nonblocking theorem yields:

Lemma. A protocol that is synchronous within one state transition is nonblocking if and only if:

(1) it contains no local state adjacent to both a commit and an abort state, and

(2) it contains no noncommittable state that is adjacent to a commit state.

State w violates both constraints of the lemma. To satisfy the lemma we can introduce a buffer state between the wait state (w) and the commit state (c). This new protocol is illustrated in figure 6. Since the new state is a committable state, both conditions of the lemma are satisfied. The buffer state can be thought of a "prepare to commit" state, and therefore, is labelled p in the illustration.

We will refer to this protocol as the canonical nonblocking protocol. It is a three phase protocol.

The above lemma is a very strong result. Since all proposed commit protocols are synchronous within one state transition, the lemma can be applied directly. In [SKEE81b] the lemma is generalized to apply to less "synchronous" protocols.

The lemma imposes constraints on the local structure of a protocol. This is convenient since it is much easier to design protocols using local constraints than using global constraints. As an example, the canonical three phase protocol was designed using the constraints

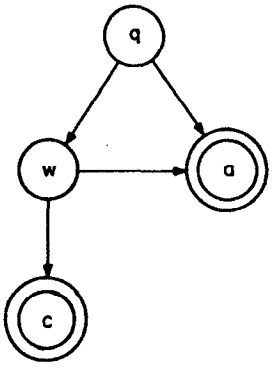

Figure 5. The canonical two phase commit protocol.

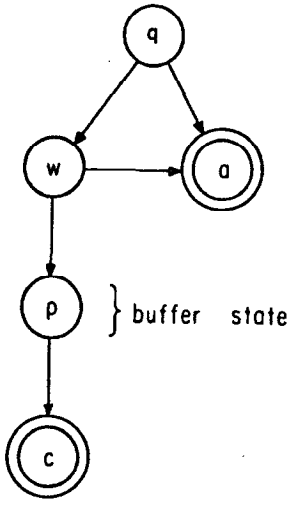

Figure 6. The canonical nonblocking commit protocol.

given in the lemma.

The significance of the canonical three phase commit protocol is that it can be specialized to yield practical nonblocking protocols. In the next sections, two nonblocking protocols are presented -- a central site protocol and a decentralized protocol. Both protocols were derived directly from the canonical three phase protocol.

A Nonblocking Central Site Protocol

A nonblocking central site protocol is illustrated in figure 7. The slave protocol is the canonical three phase protocol (with appropriate messages added). The coordinator protocol is also a three phase protocol that is a straightforward extension of the two phase coordinator protocol. The "prepare" (p) state in the coordinator directs the slaves into their corresponding "prepare" state.

A Nonblocking Decentralized Protocol

A nonblocking decentralized protocol is illustrated in figure 8. Again, the protocol is the canonical nonblocking protocol. The addition of the "prepare" state translates to another round of messages in the decentralized class.

5. Termination Protocols

Termination protocols are invoked when the occurrence of site failures render the continued execution of the commit protocol impossible. This occurs when the coordinator fails in a central site protocol, or when any site fails during a decentralized protocol. The purpose of the termination protocol is to terminate the transaction at all operational sites in a consistent manner.

Clearly, a termination protocol can accomplish its task only if the current state of at least one operational site obeys the conditions given in the fundamental nonblocking theorem. However, since subsequent site failures may occur during the termination protocol, in the worst case it will be able to terminate correctly only if all of the operational sites obey the fundamental nonblocking theorem.

We now present a central site termination protocol. It will successfully terminate the transaction as long as one site executing a nonblocking commit protocol remains operational.

A decentralized termination protocol is presented in [SKEE81b].

Central Site Termination Protocol

The basic idea of this scheme is to choose a coordinator, which we will call a backup coordinator, from the set of operational sites. The backup coordinator will complete the transaction by directing all the remaining sites toward a commit or an

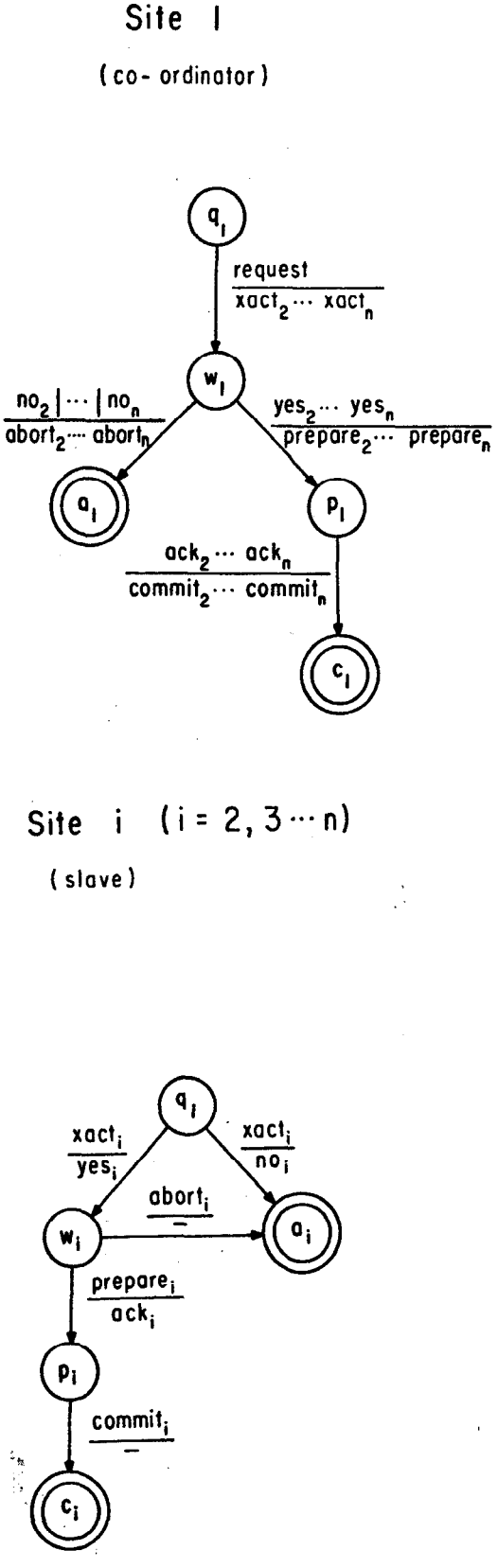

Figure 7. A (three phase) nonblocking central site commit protocol.

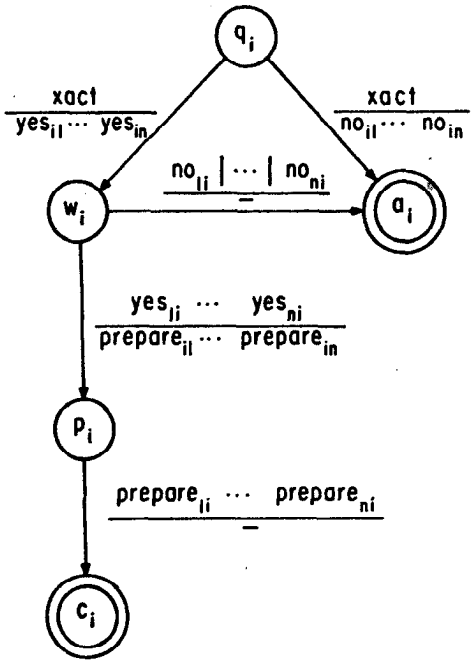

Figure 8. A (three phase) nonblocking decentralized commit protocol.

abort. Since the backup can fail before terminating the transaction, the protocol must be reentrant.

Backup coordinators were introduced in SDD-1 [HAMM79]. The scheme presented is a modification of that scheme.

When the termination protocol is invoked, a backup must be chosen. The method used is not important. The sites could vote, or alternatively, the choice could be based on a preassigned ranking.

Once the backup has been chosen, it will base the commit decision only on its local state. The rule for deciding is:

> Decision Rule For Backup Coordinators. If the concurrency set for the current state of the backup contains a commit state, then the transaction is committed. Otherwise, it is aborted.

The backup executes the following two phase protocol:

Phase 1: The backup issues a message to all sites to make a transition to its local state. The backup then waits for an acknowledgment from each site.

Phase 2: The backup issues a commit or abort message to each site. (By applying the decision rule given above.)

If the backup is initially in a commit or an abort state, then the first phase can be omitted.

Phase 1 of the backup protocol is necessary because the backup may fail. By insuring that all sites are in the same state before committing (aborting), the backup insures that subsequent backup coordinators will make the same commit decision. A proof of correctness for this protocol can be found in [SKEE81b].

Let us consider an invocation of the protocol by the canonical three phase commit protocol. The backup will chose to abort on states q, w, and a, and to commit on states p and c. If the chosen backup was in state p initially, then the messages sent to all sites are:

(1) "move to state p", and

(2) "commit".

6. Conclusion

In this paper we formally introduced the nonblocking problem and the associated terminology. Although this problem is widely recognized by practitioners in distributed crash recovery, it is the author's belief that this is the first time that the problem has been treated formally in the literature.

Also, the two most popular commit classes -- central site and decentralized -- were characterized. Every published commit protocol is a member of one of the classes. These classes are likely to prevail in the future.

We illustrated each commit class with a two phase protocol. Two phase protocols are popular because they are the simplest and the cheapest (in the number of messages) protocols that allow unilateral abort by an arbitrary site. Unfortunately, two phase protocols can block on site failures.

The major contributions of this paper are the fundamental nonblocking theorem and, from it, necessary and sufficient conditions for designing both central site and distributed nonblocking protocols.

We presented two such nonblocking protocols: the three phase central site and the three phase distributed commit protocols. The three phase protocols were derived from the two phase protocols by adding a "prepare to commit" state. This addition is the least modification that can be made to a two phase protocol in order for it to satisfy the fundamental nonblocking theorem. Therefore, such three phase protocols are the simplest (and cheapest) nonblocking protocols.

Nonetheless, an additional phase imposes a substantial overhead (in the number of messages). This overhead can be reduced by having only a few sites execute the three phase protocol; the remaining can execute the cheaper two phase protocol. The transaction will not block as long as one of the sites executing the three phase protocol remains operational. Since two site failures are always necessary to block a transaction ([SKEE81b]), the number of sites executing the three phase protocol should be greater than two.

Lastly, we presented a termination protocol to be invoked when a coordinator fails in a central site commit protocol or when any site fails in a decentralized commit protocol.

It is not necessary that the commit protocol and the termination protocol belong to the same class. In some environments, it maybe reasonable to run a central site commit protocol and a distributed termination protocol.

REFERENCES

[ALSB76] Alsberg, P. and Day, J., "A Principle for Resilient Sharing of Distributed Resources," Proc. 2nd International Conference on Software Engineering, San Francisco, Ca., October 1976.

[GRAY79] Gray, J. N., "Notes on Database Operating Systems," in Operating Systems: An Advanced Course, Springer-Verlag, 1979.

[HAMM79] Hammer, M. and Shipman, D., "Reliability Mechanisms for SDD-1: A System for Distributed Databases," Computer Corporation of America, Cambridge, Mass., July 1979.

[LAMP76] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed Storage System," Tech. Report, Computer Science Laboratory, Xerox Parc, Palo Alto, California, 1976.

[LIND79] Lindsay, B.G. et al., "Notes on Distributed Databases", IBM Research Report, no. RJ2571 (July 1979).

[LORI77] Lorie, R., "Physical Integrity in a Large Segmented Data Base," ACM Transactions on Data Base Systems, Vol. 2, No. 1, March 1977.

[ROTH77] Rothnie, J. B., Jr. and Goodman, N., "A Survey of Research and Development in Distributed Database Management," Proc. Third Int. Conf. on Very Large Databases, IEEE, 1977.

[SKEE81a] Skeen, D., "A Formal Model of Crash Recovery in a Distributed System", IEEE Transactions on Software Engineering, (to appear).

[SKEE81b] Skeen, D., "Crash Recovery in a Distributed Database System," Ph. D. Thesis, EECS Dept., University of California, Berkeley (in preparation).

[STON79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies in Distributed INGRES," IEEE Transactions on Software Engineering, May 1979.

[SCHA78] Schapiro, R. and Millstein, R., "Failure Recovery in a Distributed Database System," Proc. 1978 COMPCON Conference, September 1978.

[SVOB79] Svobodova, L., "Reliability Issues in Distributed Information Processing Systems," Proc. 9th IEEE Fault Tolerant Computing Conference, Madison, Wisc., June 1979.