CUCS-457-89

A Non-blocking Commitment Protocol

Dan Duchamp
Computer Science Department
Carnegie-Mellon University[1]

Abstract

A "non-blocking" commitment protocol is one that ensures that at least some sites of a multi-site transaction do not block in spite of any single failure. This paper describes a quorum-based non-blocking commitment protocol that also subsumes the functions of termination and recovery protocols. The protocol survives any single site crash or network partition provided that the failure is not falsely detected. The protocol is correct despite the occurrence of any number of failures, and whether or not failures are falsely detected. When there is no failure, the protocol requires three phases of message exchange between the coordinator and the subordinates and requires each site to force two log records. Read-only transactions are optimized so that a read-only subordinate typically writes no log records and exchanges only one round of messages with the coordinator. Sites can forget the transaction after it terminates everywhere. Finally, a fundamental result about quorum-based commit protocols is uncovered: they are effective only for transactions involving more than three sites.

This work was supported by IBM and the Defense Advanced Research Projects Agency, ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies or of the United States Government.

---

[1]Author's current address: Computer Science Department, Columbia University, New York, NY 10027. The former Computer Science Department at Carnegie-Mellon University is now called the School of Computer Science.

Table of Contents

1. Introduction

A distributed transaction requires a commitment protocol to ensure that all sites agree whether the transaction commits or aborts. A non-blocking commitment protocol is one that permits at least some sites to ***terminate*** (i.e., commit or abort) in spite of the occurrence of any single failure before or during execution of the protocol. The standard centralized two-phase commitment protocol [9, section 5.8.3.3] is not non-blocking because if a prepared subordinate loses contact with the coordinator (either because the coordinator crashes or because of a network failure), then the subordinate must remain prepared until the failure is repaired and communication with the coordinator is reestablished. Until then, the subordinate continues to hold write locks for the transaction, and is said to be ***blocked.***

Blocking is undesirable because if the data at the subordinate is more valuable than that at the coordinator, then the unavailability of data at the blocked subordinate may be more harmful than the unavailability of the coordinator. An example of this situation is when a data-rich mainframe serves as a subordinate to transactions initiated by user-controlled workstations. Also, if data is replicated, then data access protocols [8] can overcome a crashed site, but not a blocked site.

In the absence of failures, non-blocking commitment protocols are inherently slower than blocking protocols [6], and so non-blocking commitment is not suitable for all applications. Its main uses are:

1. For applications that are willing to sacrifice some performance in return for higher availability.
2. For large transactions. If the cost of commitment is a small part of the whole cost of a transaction, then the advantages of non-blocking commitment make it desirable.
3. For transactions executed at sites spanning a wide area. In such a configuration, subordinates remain prepared longer, and network failures are more likely.
4. For systems (such as Argus [18]) in which transaction management code runs in the same address space as user-written code. In such a system the coordinator can be expected to fail more often because user-written code is presumably more error-prone.

This paper describes a non-blocking commitment protocol that permits at least some functioning sites to terminate in spite of any single site crash or network partition. This is optimal resiliency, since Skeen has shown that no protocol can be non-blocking despite any two failures [23, pp. 83-85], and since it is likewise impossible to ensure that all sites can terminate during a single partition [21, p. 139]. The protocol uses the quorum consensus technique [8] to avoid blocking during partition, and is a centralized protocol that becomes gradually decentralized as failures occur. It can be viewed as an improved version of a similar protocol described in [2, pp. 256-260]. The improvements are:

- An optimization that, typically, allows the processing of read-only sites and read-only transactions to be as fast as with the Presumed Abort variation of two-phase commit [19].

- Addition of the ability for all sites to forget (i.e., expunge data structures pertaining to) the transaction after it is terminated. This is an important practical feature, and correctly adding this ability to a non-blocking protocol is not as obvious as it may seem. As Section 2.2 argues, a protocol that forgets too soon is incorrect, yet no existing specification of any non-blocking commitment protocol indicates how to forget.
- Design such that all actions to be taken both before and after a failure are integrated into a single protocol. Normally, [23, 2, 3] the non-blocking problem is divided into four sub-protocols:
    1. A "commit protocol" executed before the failure.
    2. A post-failure "termination protocol" executed by sites that survive the failure.
    3. A post-failure "recovery protocol" executed by crashed sites once they recover.
    4. In addition, centralized termination protocols typically make use of an additional "election protocol" for selecting a replacement coordinator.

  Having one rather than three or four protocols eases the tasks of reasoning about and implementing the protocol.

In addition, this paper presents the following advantages over previous descriptions [24, 2, 3] of quorum-based non-blocking protocols:

- Elucidation of several performance optimizations beyond the read-only optimization mentioned above.
- A complete specification that includes all knowledge needed to implement the protocol — including exactly when to write log records, and what the contents of log records and messages should be.
- Detailed arguments about the safety and liveness of the protocol.
- Analysis of normal case (i.e., failure-free) performance, and comparison with measured results of an implementation.
- Discussion of a depressing fact about quorum-based non-blocking commit protocols: they are effective only for transactions involving at least three sites. This fact is fundamental, but has never before been explicitly mentioned in several discussions of quorum-based non-blocking commit protocols [23, 24, 3, 2].

The components of this presentation are description of the protocol in Section 2, demonstration of safety and liveness in the next section, and performance analysis in Section 4. Appendix I contains one "animation" example of the protocol in operation during a failure, while Appendix II contains a complete specification of the protocol.

2. Protocol Description

2.1. Assumptions

It is assumed that when the commit protocol begins, the operations comprising the transaction are finished; i.e., the transaction is not spreading to new sites while the commit protocol executes. The only assumptions that must be made about the environment are that each site has a single stable storage log that can be written atomically, that communication is point-to-point, and that possible failures are described by a failure model:

1. Any process or site may fail at any time, but must be "fail stop," meaning that a failure results in the site or process halting immediately, then losing its volatile memory.
2. The fail stop condition implies that malicious failures [16] do not occur. Specifically, processes do not "tell lies," no network message will be undetectably altered, and the network will not spontaneously create good messages. Messages may however be lost, duplicated, or reordered. No bound on delivery time is assumed.
3. Any two sites may lose the ability to communicate, either in one or both directions, and the network may ***partition***. A partition is the separation of a completely connected network into *exactly two* completely connected subnetworks that cannot communicate with one another. The typical cause of partition is the crash of a gateway.
4. Failure is detected by timing out on an expected message. It is not possible to determine the type of failure.
5. Every failure is eventually repaired.

This model matches closely the events that do and do not happen in the real world.

Although the protocol is independent of the methods used for concurrency control and recovery, this paper is written as if the recovery method is logging, the concurrency control method is locking, and each site has a transaction controller that executes the commitment protocol and controls the dropping of locks. These conventions are adopted only in order to be precise in discussing the actions required at various times. Also, the communicating entities are called "sites" that communicate via a "network," although the protocol applies more generally to processes communicating via any sort of communication system governed by the failure model.

2.2. Operation Without Failures

In the absence of failures, the non-blocking commitment protocol has three phases. In the first "prepare phase" the coordinator tries to have all sites become prepared to either commit or abort. The purpose of the second "replication phase" is to have the coordinator replicate at subordinate sites the information that it will use to make the commit/abort decision, namely whether all sites are prepared. (Recording the *fact* that all sites are prepared is referred to as ***joining the commit group.*** Recording the *perception* that some site is not prepared is referred to as ***joining the abort group.***)

In the third "notify phase" the coordinator makes the decision and informs subordinates of the outcome. After the third phase — once all sites have committed or aborted — there is an additional seventh message, as explained below. The atomic commitment point occurs during the replication phase once enough sites have recorded the information given to them by the coordinator. As in two-phase commit, all sites retain the ability to "abort unilaterally" before they prepare. The first and third phases of this protocol correspond closely to the two phases of two-phase commit. Beyond having three centralized phases, the other salient aspects of the protocol are that a subordinate does not forever await messages from the coordinator but instead times out and itself becomes another coordinator, and that a crashed site assumes the role of a coordinator when it recovers.

The precise steps of the protocol for an update transaction are:

1. The coordinator prepares by dropping its read locks and forcing a *prepare log record* that contains the list of its write locks, the quorum sizes and the list of subordinates.

2. The coordinator sends *prepare* asynchronously to all subordinates, then awaits responses. The prepare message must contain enough information to allow the subordinate to later behave as a coordinator: the list of sites involved in the transaction and the two quorums sizes.

3. Each subordinate drops its read locks and attempts to prepare. To prepare, the subordinate forces a *prepare log record* that contains its write locks, the quorum sizes and the list of all sites. It then sends *prepare-ack (yes)* to the coordinator. If the site cannot prepare, it sends *prepare-ack (no)*, undoes its updates, drops its write locks, and spools an abort record. To "spool" a log record means to write it into a log buffer; the next log force will write the entire buffer into the log.

4. The coordinator waits until either all *prepare-ack* messages are received or a timeout occurs. If all sites are prepared, then the coordinator forces an *in-group log record*. The record should contain the state of all sites, the state of the coordinator being in-group/commit. It then sends *join-group (commit)*, which also contains state of all sites as known to the coordinator. If some site is unprepared or a timeout occurred, then the coordinator forces the *in-group log record* with the coordinator state being in-group/abort. It then sends *join-group (abort)* to all subordinates. This message also lists every site and its state.

5. A subordinate that receives the *join-group* message joins the specified group provided it is not already in a group. It does so by forcing an *in-group log record* that contains the list of all sites and their states. It then sends *in-group* back to the coordinator. If the subordinate times out waiting for *join-group*, then it becomes a coordinator in the prepared state.

6. The coordinator collects *in-group* messages, checking if a quorum is formed. While no quorum exists, the coordinator must continue to resend *join-group* to those subordinate sites not yet in a group. Once a quorum exists, the coordinator either commits or aborts. Committing is easy: the coordinator forces an *outcome log record* that specifies commit. To abort, updates are undone, write locks are dropped, and then an *outcome log record* specifying abort is spooled. After committing or

aborting, the coordinator sends *outcome* to all subordinates. The *outcome* message and log record need not include any extra information beyond that which indicates whether to commit or abort.

7. A subordinate that receives *outcome* commits or aborts just as the coordinator did. It then replies with *outcome-ack*. As at the coordinator, the log record need not include extra information. If the subordinate times out waiting for the outcome, it becomes a coordinator in the appropriate in-group state.

8. The coordinator resends *outcome* until it receives *outcome-ack* from all subordinates. It then sends *forget* to all sites, and forgets the transaction. Forgetting is accomplished by spooling a *done record* into the log buffer. The purpose of a done record is to indicate that this transaction's portion of the log can be reclaimed.

9. A subordinate that receives *forgets* forgets in the same way the coordinator does. If it times out, it becomes a coordinator and sends *outcome* to all sites, which will reply with *outcome-ack*.

Clearly, the coordinator should retransmit unacknowledged messages a few times to prevent needless subordinate timeouts. Beyond this, several important optimizations are possible, as explained in Section 2.5.

Figure 2-1 displays the messages sent between coordinator and subordinate when a transaction commits without failure. The three phases are obvious. The extra message is necessary if sites are to be able to forget, according to three steps of reasoning. First, a site with no record of a transaction can join the abort group. Second, it is possible for one site to be committed while another is still trying to gather an abort quorum. So if a subordinate were to commit and then forget, it could join the abort group after having forgotten. In this fashion, a single site could join both the commit group and, later, the abort group, and allow the same transaction to commit at some sites and abort at others. To ensure correctness, *no* site forgets until *all* sites (including those that may have crashed or been isolated by partition) have acknowledged the outcome. While forgetting thus can be delayed by a failure, this is not blocking, since locks have been dropped.

2.3. Replication Phase

The extra replication phase is necessary in order to have a non-blocking protocol that operates within a failure model that allows partitions. The foundation upon which the correctness of the protocol rests is that (for the purpose of terminating a particular transaction) a site can join only one group, ever. At any given time, some coordinators may be increasing the membership of the commit group while other coordinators are increasing the membership of the abort group. The outcome is determined by which group succeeds in gathering the required number of sites.

The requirement for an exclusive outcome is set by the well-known quorum consensus method: commit (abort) can take place provided that the commit (abort) group consists of a commit (abort) ***quorum***. If C is the number of sites required for a commit

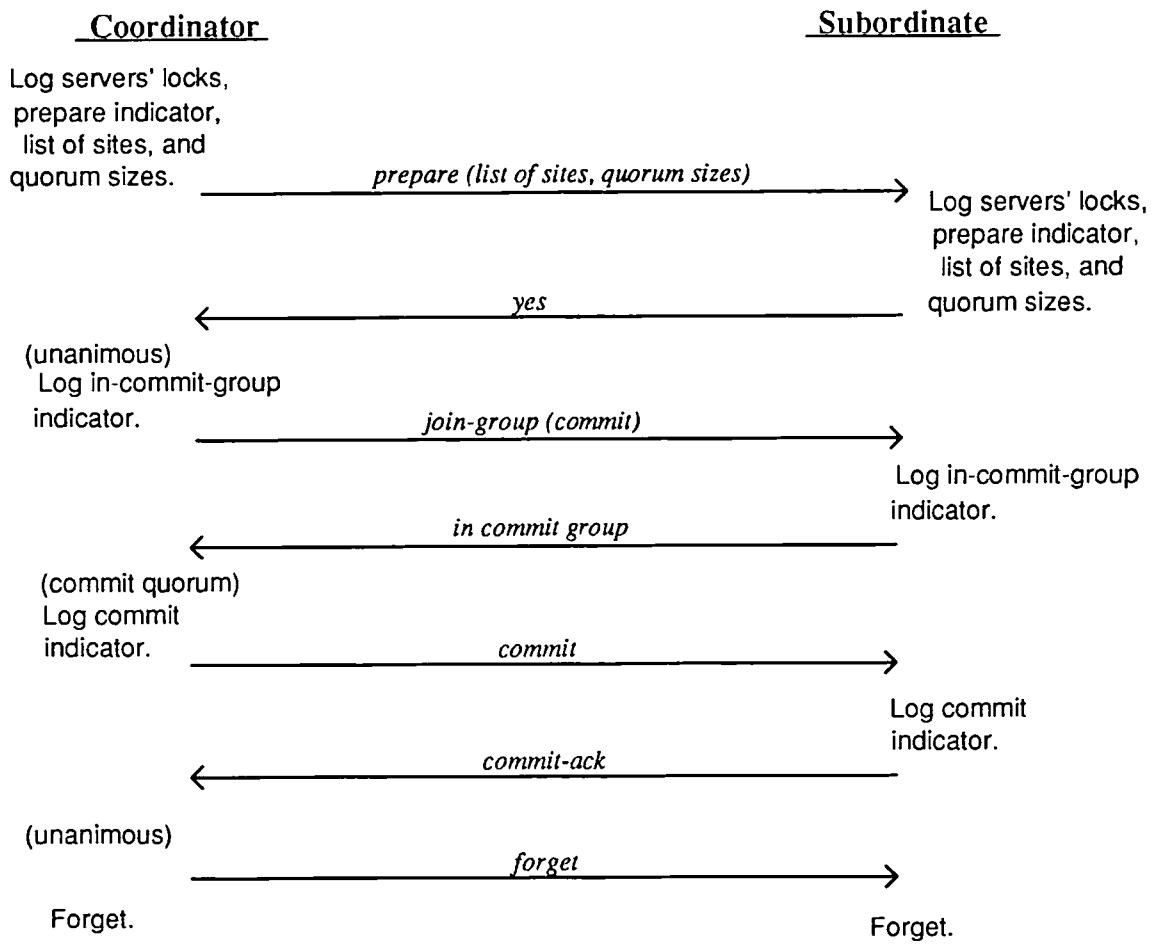

Figure 2-1: Non-blocking Commitment without Errors

The typical sequence of messages exchanged between a coordinator and a subordinate when a transaction commits without error. The notation "(commit quorum) Log commit indicator" means that once a commit quorum of sites have responded saying that they are in the commit group, write the commit indicator into the log.

Commands to join the commit group or to join the abort group are not separate message types. Instead, there is a single join-group message, and whether to join the commit or abort group is passed as data.

quorum, A is the number for an abort quorum, and N is the total number of sites, then the correct relation between the quorums is:

$$C + A = N + 1, \qquad C, A < N.$$

This relation is derived from the essential requirements for a non-blocking protocol. Mutual exclusion of outcomes requires that

$$N < C + A. \tag{1}$$

Also, it must be possible to reach either outcome even if the other outcome is as little

as one site short of achieving a quorum. Translated into mathematics, this is

$$N - (\min(C,A) - 1) \geq \max(C,A),$$

and

$$N - (\max(C,A) - 1) \geq \min(C,A).$$

These inequalities are in fact both the same, namely

$$N + 1 \geq \max(C,A) + \min(C,A) \tag{2}$$

$$N + 1 \geq C + A$$

Together, Inequalities 1 and 2 determine the equation

$$C + A = N + 1. \tag{3}$$

The condition

$$C, A < N \tag{4}$$

reflects the fact that a quorum-based protocol cannot tolerate a single site crash if all N sites are required to reach an outcome.

An unfortunate consequence of Inequality 4 is that quorum-based protocols are useful only for transactions with three or more sites: if N is only 2, then by Equation 3, either C or A must also be 2. In this case, if one site were to join the group whose quorum was 2 and then the other site crashed, then that site would be blocked.

2.4. Handling Failures

2.4.1. Coordinator Loses a Subordinate

If the coordinator loses contact with some subordinate, its behavior in the non-blocking protocol is similar to that in two-phase commit: timeout and abort if waiting for *prepare-response*, resend *outcome* if waiting for *outcome-ack*. The only difference is that the non-blocking protocol will block if too many subordinates crash before or during the replication phase: the coordinator may not be able to gather a quorum for either commit or abort. In this sense, non-blocking commitment is actually *less* resilient to *subordinate crashes* than two-phase commitment; with two-phase commitment, the coordinator may terminate no matter how many subordinates crash. This resiliency is traded in return for being able to tolerate the loss of the coordinator.

2.4.2. Subordinate Loses the Coordinator

A subordinate may become (another) coordinator if it times out waiting for a message from the original coordinator; communication with the other sites is always possible because each subordinate receives the list of sites in the first message of the protocol. The correctness of the protocol depends in no way upon the length of the timeout interval, when subordinates become coordinators, or how many do so. It is not necessary to execute an election protocol [7]. One may think of the subordinates as making arbitrary decisions about whether and when to become coordinators.

Coordinators and subordinates pass through the same states, so the protocol is

symmetric. The progression of states is shown in Figure 2-2. A subordinate that becomes a new coordinator executes the protocol just as the original coordinator would in the same state. The new coordinator first sends to all other sites the last message it received, to ensure that all sites are at least as advanced as it is.[2] Commitment then proceeds normally, although perhaps with more than one coordinator.

Figure 2-2: Progression of State Transitions of Non-blocking Commitment Protocol

Every site travels one path through the DAG. Of course, either all sites commit or all abort. Notice that a site may join the commit (abort) group, but then abort (commit). The group that it joined failed to gain a quorum, while the other group succeeded.

2.4.3. Recovery from Crash

After recovering from a crash, a site becomes a coordinator. This policy obviates the question of which site will be coordinator if all sites crash, perhaps due to a power failure. If the recovered site had not yet terminated before crash, then it is certain that at least one other site will be able to give it the outcome: the coordinator, which has not yet forgotten. If the recovered site had terminated before crash, then it will

[2]The "advancedness" of a state is its level in the state transition graph of Figure 2-2.

begin by sending the *outcome* message. Even if, as is likely, all other sites have terminated and possibly also forgotten, they will satisfy the newly recovered site by sending *outcome-ack* (see Table II-11).

Becoming a coordinator after recovery ensures that the transition from subordinate to coordinator is permanent. Furthermore, the opposite transition (coordinator to subordinate) never takes place.

2.4.4. Recovery from Partition

If a partition lasts longer than some subordinate's timeout interval, then at least one coordinator will operate within each subnetwork. So at least one subnetwork will terminate but not forget. The coordinator(s) in that subnetwork will continue to send *outcome* until the partition is repaired and the sites that formed the other subnetwork reply with *outcome-ack*. (Obviously, a coordinator should increase the interval between consecutive sends up to some reasonable limit.)

2.4.5. Dueling Coordinators

The presence of multiple simultaneous coordinators is referred to as ***dueling***. To a subordinate, the existence of dueling coordinators seems like delayed or duplicate messages, a circumstance that must be handled anyway. However, it is potentially confusing for a coordinator to receive the same types of messages it is sending. The rule for how one coordinator reacts to the messages sent by another is based on the relative states of the two sites.

If the receiving coordinator is in a *less advanced* state than the sender, then it should do as instructed, respond as if it were a subordinate, but remain a coordinator. For example, a prepared coordinator that is told to commit should commit, then send *outcome-ack* to the other coordinator; next, it should send *outcome (commit)* to all subordinates. If the receiving coordinator is in a *more advanced* state, then it should ignore the message and continue treating the sender as a subordinate. Lastly, if both coordinators are in the same state, then the receiver should respond as if it were a subordinate. So if a committed coordinator receives *outcome (commit)*, it should respond with *outcome-ack*.

The major benefit of tolerating multiple simultaneous coordinators is that the protocol is not burdened with the complication of requiring that every subordinate agree upon which site is the coordinator. A minor benefit is an increased ability to terminate in pathological cases where the communication network is very disconnected, as explained in Section 2.5.1.

The list of sites is carried in the *prepare* message. How long a subordinate waits before timing out should be made proportional to its position within the list of sites. This lessens dueling. Lessening dueling is desirable in order to reduce network activity during a failure, but is not needed for correctness.

2.4.6. Performance/Availability Tradeoff: Choosing Quorums

Two factors affect the behavior of the protocol in the (normal) case when a transaction commits without failure, and in the case when a partition occurs. They are the relative sizes of the two quorums and the order in which join-group messages are sent.

In order to speed the normal case, the best choice of quorums is C = 2, A = N-1. If one subordinate is more likely to respond quickly to the join-group message, then that site should be the first destination whenever the coordinator sends join-group messages. Such a policy increases the speed with which the commit group achieves a quorum.

Selecting quorum sizes to minimize blocking when a partition occurs requires knowing or assuming the failure probabilities of the components of the system. If certain partitions are more likely than others, then the chance of blocking can be reduced by sending the first join-group message to a site that is more likely to be in the partition not containing the coordinator. Unfortunately, it is unlikely that quorums can be tuned to improve both the no-failure case and the expected-partition failure case. Partitions typically occur because of the crash of a gateway. Also, communication between two sites separated by a gateway is usually relatively slower than communication between two sites on the same network. To tune for speed in the normal case, one wants to use one of the faster (i.e., common network) communication paths. To tune for availability in the expected-partition case, one wants to use one of the slower (i.e., separated by a gateway) communication paths.

2.5. Optimizations

2.5.1. Availability Optimzation: Information Accumulation

Whenever any of the first four messages (*prepare*, *prepare response*, *join-group*, *in-group*) is sent, it should contain the state of every site as known to the sender. This feature increases the number of sites reaching an outcome when the communication network is less than completely connected, by using information about other sites that is relayed indirectly. For example, suppose that after several partitions occur, two subnetworks are connected by only a one-way link between one site in each. Even if each subnetwork is not large enough to terminate by itself, one may be able to terminate using information passed across the link about the states of sites in the other subnetwork. Because of the linear progression of states within the protocol, messages may contain information that is old and possibly useless, but never wrong.

2.5.2. Performance Optimization: Delaying Messages

The last two messages of the protocol, *outcome-ack* and *forget*, serve only to inform sites when they can safely forget the transaction. Accordingly, these messages should be delayed for some time in case they can be piggybacked on other messages between the same two sites.

2.5.3. Performance Optimization: Read-only Sites

It is common for individual processes, sites, or entire transactions to be read-only. In two-phase commitment, a read-only site requires fewer messages and writes no log records. This property can often but not always be preserved by non-blocking commitment.

When asked to prepare, a read-only site drops its locks, votes read-only, but retains its memory of the transaction. (In two-phase commitment, a subordinate that votes read-only may forget immediately after voting.) With non-blocking commitment it is necessary for the transaction manager of a read-only site to remember the transaction because it may be asked later to join the commit group.

If the coordinator sees that the transaction is completely read-only, then it next sends *forget*. If there is a mix of read-only and update sites, then commitment proceeds normally. The coordinator invites subordinates to join the commit group. If there are enough update sites to form a commit quorum, read-only sites should not be asked to join. If it is necessary to ask some read-only sites to join the commit group, then they write the in-group record directly without writing a prepare record. Although they participated in the second phase, these sites need not be included in the third phase. The purpose of informing an update site that a transaction has committed (or aborted) is to allow it to drop its locks. Read-only sites have already dropped their locks. In summary, read-only sites should be left out of the replication phase if possible, and need never be involved in the notify phase. They must be told to forget, however.

Having read-only sites not write a prepare record means that it must be possible for a site that has no memory of a transaction to join a group. For example, a read-only site may crash and then later be asked to join a group. The rule for which group a site should join if it lacks memory of the transaction is:

- No other site in commit group: join abort group. It is not certain that all sites prepared.
- Commit and abort groups are same (non-zero) size: join commit group, to help commit rather than abort.
- One group larger than other: join the larger group, to help terminate as soon as possible.

2.5.4. Performance Optimization: Eliminating Log Forces

Careful analysis reveals that two log writes need not be log forces. First, the commit/abort record at a subordinate need not be forced, as explained in [4, pp. 50-52]. Second, it is necessary to force only one of the coordinator's first two records (prepared or in-group).

If the prepare record *is not* forced, then the transaction must abort if the coordinator crashes; the coordinator's prepare record is placed in the log when it forces the in-group record. If the prepare record *is* forced, then the in-group record need not be: if the subordinates already constitute a quorum or are only one site shy, then the

coordinator can force a combination in-group and commit/abort record.

The latter is preferable. In addition to allowing a transaction to commit in spite of a failure, there is an availability advantage in multiple-failure cases. Provided that the coordinator has not already joined a group, it can examine the number of sites in each group, and perhaps force an outcome by "casting the deciding vote." For example, if the abort group is only one site shy of a quorum, it is preferable for the coordinator to join the abort group in order to terminate the transaction even if it was soliciting sites to join the commit group.

Having a coordinator delay joining a group should be handled carefully. The danger is that a "procrastinating" coordinator could unnecessarily prevent a transaction from terminating. For example, consider a partitioned subnetwork of M sites consisting of two coordinators and the rest subordinates. Suppose also that the quorum for committing is M, and that every subordinate has joined the commit group, but that the two coordinators are still only prepared. The protocol must be designed so that the two coordinators do not forever send each other the join-group message, each waiting for the other to become the M-1st site in the commit group.

Accordingly, the rule for how a coordinator reacts to a message from another coordinator in the same state should be amended to: if the receiving coordinator is ranked lower than the sender in the list of sites, it will react as if it were in a less advanced state. This rule establishes a strict ordering of the relative states of two coordinators.

3. Informal Correctness Arguments

There are two salient issues to investigate: safety and liveness. Showing safety requires demonstrating that never does some site commit and another abort. Showing single-failure liveness requires demonstrating — that provided that only one site crash or network partition occurs during the execution of the protocol — that some site is able to terminate without waiting beyond the specified timeout intervals, and that blocked sites become unblocked once the failure is repaired.

3.1. Safety

The safety of the protocol rests on the fact that a site's membership in a group is stable and therefore a quorum is also stable. Once a site joins a group it remains in the group (and refuses to also join the other group) until after it is known that all sites have reached the outcome and there will be no further attempts to form groups for that transaction. Provided that quorum sizes are chosen according to Equation 3, it is impossible that the commit group and the abort group can both reach a quorum any time during the execution of the protocol. This is the crux of all protocols based on the quorum consensus method.

This simple and intuitive argument is developed more fully and is connected more precisely to the specification of the protocol by the 13-step chain of reasoning enumerated below. Statements in normal typeface are true statements about the protocol.[3] Statements in italic typeface are deductions based on previous statements.

1. It is assumed that when the protocol starts, all operations have terminated or have been aborted (i.e., the transaction is quiescent), and every site that is involved is known to the original coordinator. The list of involved sites is transmitted in the first message and is placed in the first log record (prepare for write sites, in-group for read-only sites), and so is always known to every site that ever becomes a coordinator.
2. A site commits or aborts only when it has positive information that a quorum has been gathered. A site may receive positive information either by receiving a message that indicates that some other site is already terminated, or — when acting as a coordinator — by receiving a message that indicates that a quorum has been formed.
3. *Therefore, to show safety, it must be shown that conflicting quorums are impossible.*
4. A site joins a group only when it is prepared or when the transaction is unknown to it.
5. Because recovery from site crash restores the state indicated by the last log record, a site remembers if it previously joined a group.
6. Once in a group or terminated, a site refuses to join the other group.
7. *Therefore, once in a group, a site remains in only that group until told to forget the transaction.*

[3] An energetic reader can verify each statement by inspecting Tables II-1 through II-11.

8. No site will send *forget* until all sites have terminated.
9. A site that is terminated will never try to form a quorum, so there will be no attempt to form a quorum after any site has forgotten.
10. *Therefore, a site joins at most one group, ever — before or after forgetting.* The lifecycle of a site is as follows: it joins a group once, then remembers that it joined until such time as it will not be asked to join the opposite group, then it forgets only after knowing that it will never again be asked to join a group for that transaction.
11. *Therefore, it is not possible for any site to join conflicting groups.*
12. *Therefore, at most quorum one can ever exist, assuming* $C + A > N$.
13. *Therefore, it is not possible for any site to decide commit/abort opposite of what another site decides.*

3.2. Liveness

The elegance of the "when in doubt, become a coordinator" policy carries a price, expressed by the "true detection assumption."

3.2.1. True Detection Assumption

For this protocol to always permit one subnetwork of a partition to terminate, there must be no false error detection. Assuming otherwise, a scenario such as this could occur:

1. The original coordinator prepares all sites and begins the replication phase. Some sites join the commit group.
2. Meanwhile, a prepared subordinate "imagines" (i.e., incorrectly detects) that the coordinator has failed. The subordinate becomes a coordinator and sends prepare messages.
3. The *prepares* of the second coordinator are lost (through bad luck), so the second coordinator begins forming the abort group. Some other prepared sites join the abort group at the behest of the second coordinator.
4. A true partition happens (the first "real" failure), leaving each subnetwork with some sites in the commit group and some in the abort group.
5. For certain choices of quorum sizes, neither subnetwork of sites will be able to form a quorum. For example, suppose N = 8, C = 5, and A = 4. If the sites are partitioned into two equal subnetworks and each subnetwork has one site in the commit group and one in the abort group, then neither subnetwork can terminate.

A second coordinator acting as an adversary can prevent quorums from forming by always doing "just the wrong thing:" getting sites on either side of the soon-to-be partition to join the opposing group before a true partition occurs.

The non-blocking protocol can be shown to be live despite a single failure only if there are no false detections of failures. Put another way, the protocol is live in the presence of any single diagnosed failure, whether the diagnosis is correct or not. The liveness argument in the next section depends upon this *true detection assumption:* every detected failure is a real failure.

3.2.2. Single-failure Liveness

Unlike the demonstration of safety, which is based on general statements, the demonstration of liveness is done with a simple case analysis.

It is clear from examination of the specification in Sections II.1 and II.2 that the state transition graph is indeed the DAG shown in Figure 2-2, so no site ever "retreats" through the states. It remains to show that not all sites "park" in any state unless there are too many failures, and that — for those sites that do park — when the failures are repaired the outcome is eventually reached.

Analysis of a subordinate is trivial. In each state, a subordinate will either be "pushed" into a more advanced state by a message from some coordinator or timeout waiting for such a message and become (permanently) a coordinator itself. Therefore, showing the liveness of a subordinate reduces to showing the liveness of a coordinator.

A coordinator will be pushed into a more advanced state by another more advanced coordinator or will force itself into the next state provided it can receive enough responses from subordinates and other coordinators, in which case it "pulls" itself forward to the next state. When a message is received from another coordinator, if the sender is less advanced the message is ignored; otherwise, the receiving coordinator is pushed. When a message is received from a subordinate, either it is a delayed response (in which case it is ignored), or it is an expected acknowledgement. In the latter case, the message contributes to the coordinator pulling itself into the next state.

A coordinator might stop advancing only when there are enough failures to prevent its pulling itself to the next state. If the coordinator fails to receive enough prepare responses, it times out and advances to the next state. If it fails to receive enough in-group responses, it blocks. If it fails to receive enough outcome acknowledgements, it continually resends *outcome*. However, this is not blocking; a site is blocked only when it is a coordinator awaiting in-group messages. Blocking cannot be caused by the crash of any single site provided that both quorum sizes are less than the number of sites, so consider the occurrence of a single partition.

What situations may exist when a coordinator is trying to form a quorum and a partition has occurred? The possible cases are:

1. Both subnetworks have all sites prepared.
2. One subnetwork has some sites in one group, while the other has all sites prepared.
3. Both subnetworks have some sites in one group. The group is the same in both subnetworks.

The true detection assumption rules out the possibility of the existence of sites in both groups before the failure. At most one kind of group may exist within a subnetwork before the failure.

Information that another site is in a more advanced state always pushes a coordinator into that state: for example, if a prepare response from a subordinate indicates that the subordinate really is in the commit group, then the receiving coordinator will use the information to conclude that *all* sites are prepared, and will begin forming the commit group. Because of this property, a subnetwork consisting of some prepared sites and some sites in a group will — barring further failures — have all sites join that group, no matter which sites act as coordinators. Therefore, Case 3 above will terminate in at least one subnetwork, Case 2 will either terminate in the advanced subnetwork (if it is big enough) or abort in the other subnetwork, and Case 1 will abort in at least one subnetwork. Blocked sites become unblocked and crashed sites recover as described in Sections 2.4.4 and 2.4.3, respectively.

4. Performance

This section examines the performance of the protocol in the case where no failures occur. Since failures are rare, this is the most important measure of commit protocol performance. For judging the latency of the normal case, two events are important: the moment at which all locks have been dropped, and the moment when the (synchronous) commit call returns to the caller. The ***critical path*** of a commitment protocol is the shortest sequence of actions that must be done sequentially before all locks are dropped and the call returns, and this measure is the focus of performance evaluation.

Experience shows that the length of the critical path is dominated by inter-site messages and log forces [5]. Because coordinator-subordinate communication is asynchronous, subordinate log forces take place roughly in parallel, and so the critical path of an N-subordinate transaction should be not greatly more than that of a 1-subordinate transaction.

The optimizations of Section 2.5.4 piggyback the non-critical messages and log forces of one transaction onto other messages and log forces, and so reduce the critical path of an update transaction to 4 log forces and 5 messages. This compares to 2 and 3, respectively, for two-phase commit. The log forces and messages in the critical path are:

- Log forces: coordinator prepare, subordinate prepare, subordinate in-group, and coordinator in-group and commit. The last two are batched together, as explained in Section 2.5.4.
- Messages: prepare, prepare response, join-group, in-group, and outcome.

The ratios of these dominant operations are 4/2 and 5/3, implying that the length of the critical path of non-blocking commit is about twice that of two-phase commit. For read-only transactions, optimizations reduce the non-blocking critical path to that of optimized two-phase commit. Comparison of the two protocols is shown in Table 4-1.

TRANSACTION TYPE	MESSAGES	LOG FORCES	MSG DELAYS	LF DELAYS
NBC update	5N	2+2N	5	4
2PC update	3N	1+N	3	2
NBC read	2N	0	2	0
2PC read	2N	0	2	0

Table 4-1: Performance Comparison of Non-blocking and Two-phase Commit Protocols

N is the number of subordinates. NBC stands for non-blocking commit, while 2PC stands for two-phase commit. Both protocols are fully optimized.

4.1. Measured Results

This non-blocking protocol has been implemented within the Camelot transaction processing facility [25]. Performance measurements were gathered on several IBM RT PCs, model 125, a 2-MIP machine. The network was a 4Mb token ring without gateways. The computers were running Mach [1], version 2.0.

The timing experiment consisted of executing a minimal distributed transaction on a coordinator and on 1, 2, and 3 subordinate sites. The "minimal transaction" performed one small operation at each site. A minimal transaction was used in order to more easily subtract the latency due to operation processing, leaving just that associated with the commitment protocol.

Table 4-2 shows the failure-free performance of the non-blocking protocol and two-phase protocol for update and read transactions, respectively. The number is not directly measured, but is necessarily derived by measuring the latency of entire transactions and then subtracting the known times of the operations outside the commit protocol.

TRANSACTION TYPE	(DERIVED PROTOCOL LATENCY)
NBC update, 2 sub.	(157.0)
2PC update, 2 sub.	(88.0)
NBC update, 3 sub.	(193.5)
2PC update, 3 sub.	(102.5)
NBC read, 2 sub.	(58.0)
2PC read, 2 sub.	(34.0)
NBC read, 3 sub.	(67.5)
2PC read, 3 sub.	(40.5)

Table 4-2: Derived Commit Protocol Latency

The cost of non-blocking commitment relative to two-phase commitment seems somewhat less than twice as high, the ratio varying from 1.88 to 1.66. This result is in agreement with the analysis above. The advantage gained by optimizing read-only transactions is clearly substantial.

In absolute terms, the protocol executes in a fraction of a second in the test environment. In order for the latency of the commitment protocol to be negligible (say, less than 5%), non-blocking commitment should be used with transactions that last longer than a few seconds. This implies that non-blocking commitment is suitable for transactions used in application programming, but not in system programming.

5. Related Work

The motivation for a non-blocking commitment protocol has existed for some time, and many protocols have been invented. The seminal work is Dale Skeen's 1982 Ph. D. dissertation [23] devoted entirely to the topic of non-blocking protocols. In addition to proving fundamental results and defining the notation that is now standard, Skeen described several protocols. The two most interesting ones are the "canonical" three-phase non-blocking commitment protocol [22] which does not survive partition, and the "quorum-based" protocol [24] which does.

The problem of finding a non-blocking commit protocol is substantially easier if the possibility of network failures is ruled out. The implication that the lack of response from a remote site means that the site has stopped is an important tool for the protocol designer. Further examples of protocols that survive only site crash failures are the four-phase protocol of the SDD-1 project [10], Le Lann's Cooperative Protocol [17, p. 46], and Yuan and Jalote's protocols for recovering from site failure with minimal message exchange [27].

The essential technique for surviving partitions is quorum consensus, and use of this technique has a long history. The protocol described in this paper is quite similar to that in [2, pp. 256-260], which is in turn similar to Skeen's decentralized quorum-based termination protocol [23, pp. 164-176], which itself makes use of the special-purpose commitment scheme outlined by Reed [20, pp. 115-118].

Because the inherent slowness of non-blocking commit is unattractive, several ad-hoc methods have been devised either to unblock a blocked transaction, or to make blocking less likely. IBM's LU 6.2 supports "heuristic commit," which allows for resolution of a blocked transaction by either manual or programmed means [12, p. 5.3-16]. The R* prototype likewise allows manual resolution of a blocked transaction [19]. Quicksilver implements two techniques for reducing the likelihood of blocking: "coordinator migration" and "coordinator replication" [11]. A two-site transaction uses coordinator migration. Coordinator migration allows the subordinate and the coordinator to switch roles, for times when the subordinate is supposed to be more reliable than the coordinator. For transactions involving more than two sites is Quicksilver uses coordinator replication. In essence, the coordinator nominates one of the subordinates to serve as a co-coordinator. Each co-coordinator coordinates commitment of approximately half the sites. If either co-coordinator fails, then the other takes over as coordinator for all sites. The advantage of coordinator replication is that it has a shortened "window of vulnerability:" the time during which some subordinates are prepared and there is a single coordinator.

Optimality work on the non-blocking problem includes Dwork and Skeen's proofs of lower bounds on the number of messages and message phases [6], study of site-optimal termination protocols [3] and message-optimal recovery protocols [27], and a series of papers [13, 14, 15, 26] that develop message-optimal commit protocols.

6. Summary

This paper describes a quorum-based non-blocking commitment protocol that also subsumes the functions of termination and recovery. Like all such protocols, it is useful only for transactions involving three or more sites. In the normal case, the protocol requires three message phases, including two log forces at each site and five messages in the critical path. In the case of failures, the protocol is non-blocking for any single site-crash or network failure provided that there is no false detection. It is correct provided that the quorum selection rules are obeyed, even if failures are falsely detected. Read-only sites need never participate in the notify phase, and often need not participate in either the replication or notify phases. A transaction that is completely read-only has the same critical path performance as optimized two-phase commitment. Sites can forget the transaction after the transaction is terminated everywhere.

I. Example Operation With Failures

Figures I-1 through I-9 display the sequence of events that take place when a partition splits five sites into two subnetworks. A new coordinator takes control within the subnetwork that is separated from the original coordinator. In this example the non-blocking protocol achieves no more unblocked sites than would be achieved by two-phase commit. The point is to illustrate the manner in which a new coordinator operates.

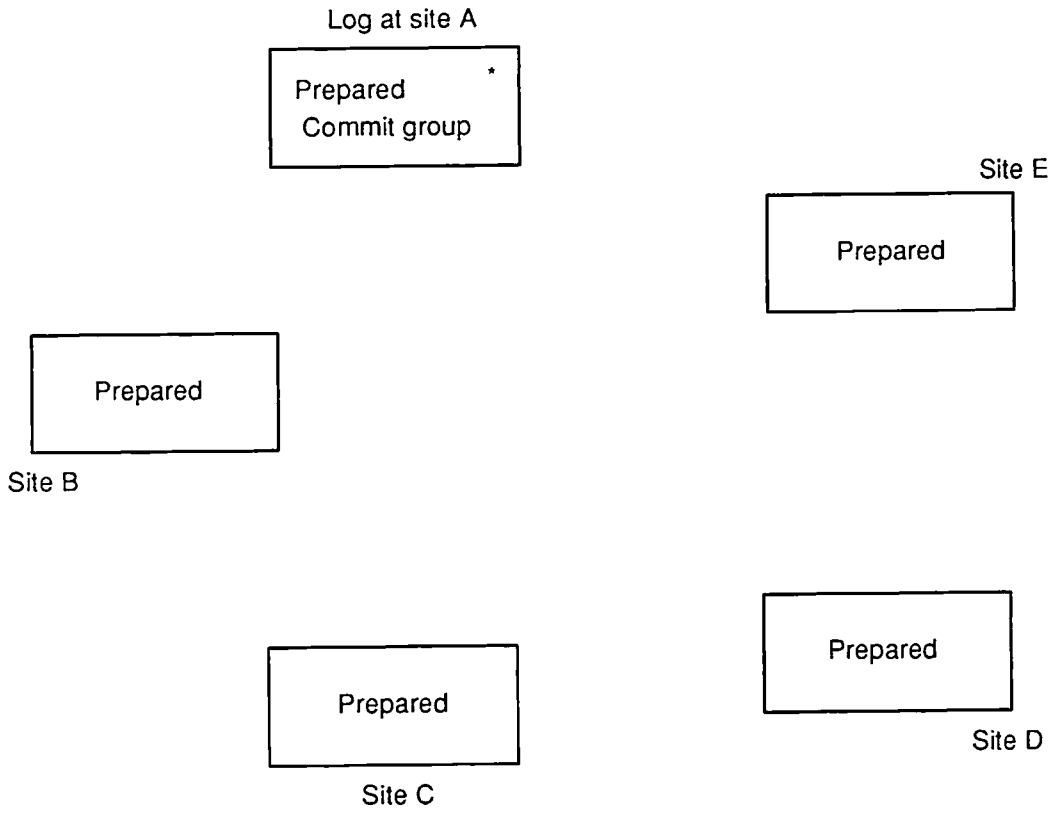

Figure I-1: Partition Example: Frame 1

The box representing a site indicates which records are in the log at that site. In the initial state of the example, every site is prepared and is aware of the quorum sizes. Three sites are required for both a commit or an abort quorum. The coordinator (denoted by the asterisk) knows that every subordinate is prepared, and has joined the commit group.

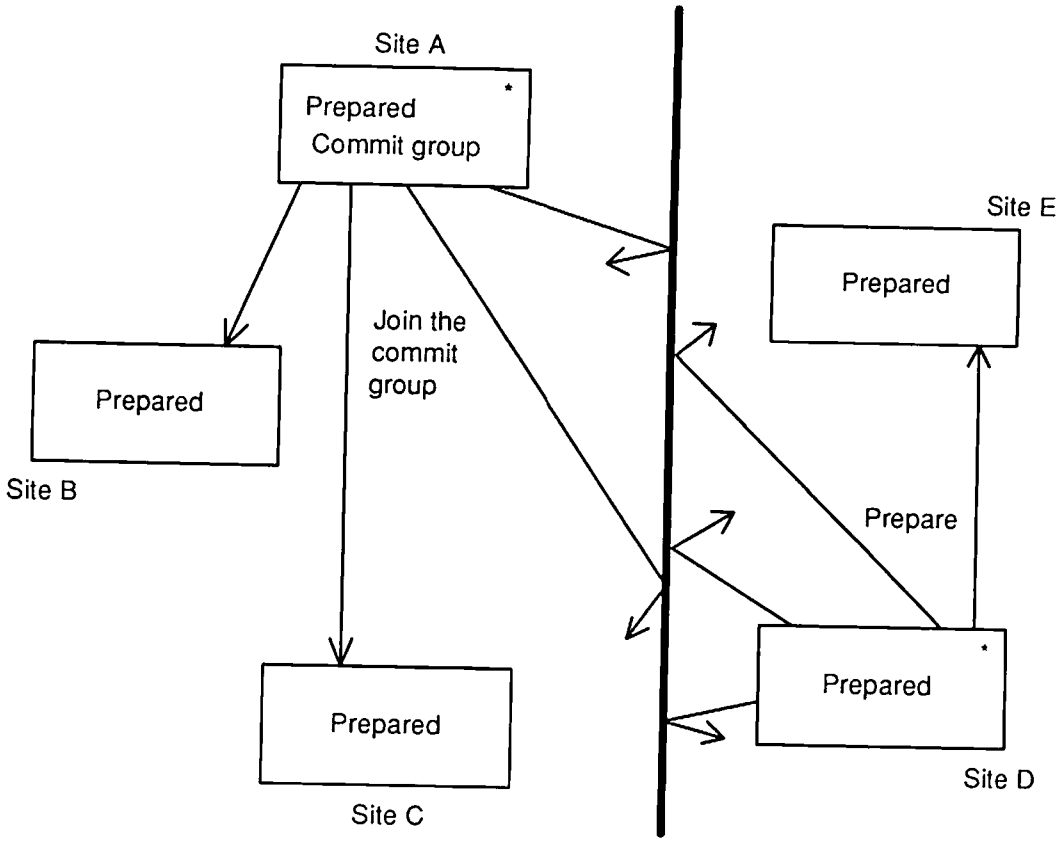

Figure I-2: Partition Example: Frame 2

The coordinator sends "join-commit-group" messages to all subordinates. A partition prevents two messages from arriving. Meanwhile, site D becomes impatient waiting for the message and converts into another coordinator. It must try to push every site into its state (prepared), so it sends prepare messages to all other sites. Again, some of the messages do not arrive because of the partition.

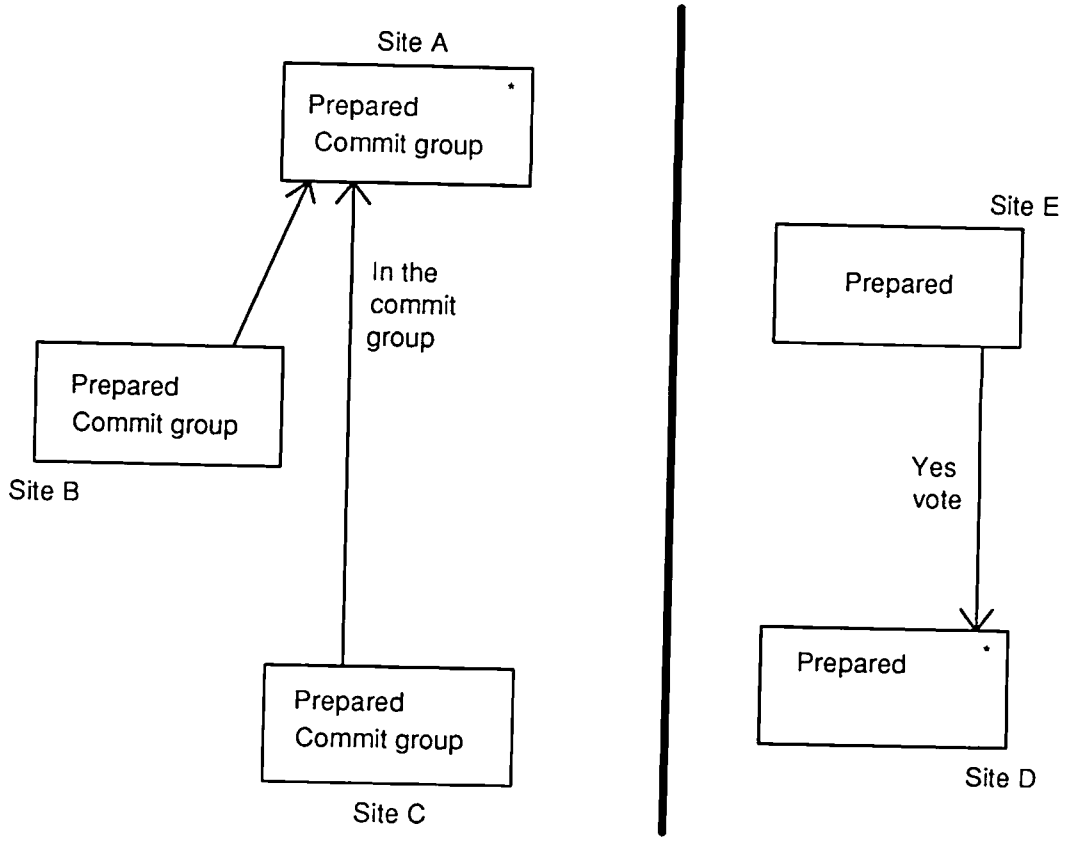

Figure I-3: Partition Example: Frame 3

Each remaining subordinate (B, C, and E) responds to whatever message it received. In the case of site E, the prepare message from Site D is a duplicate. Site E reiterates its vote.

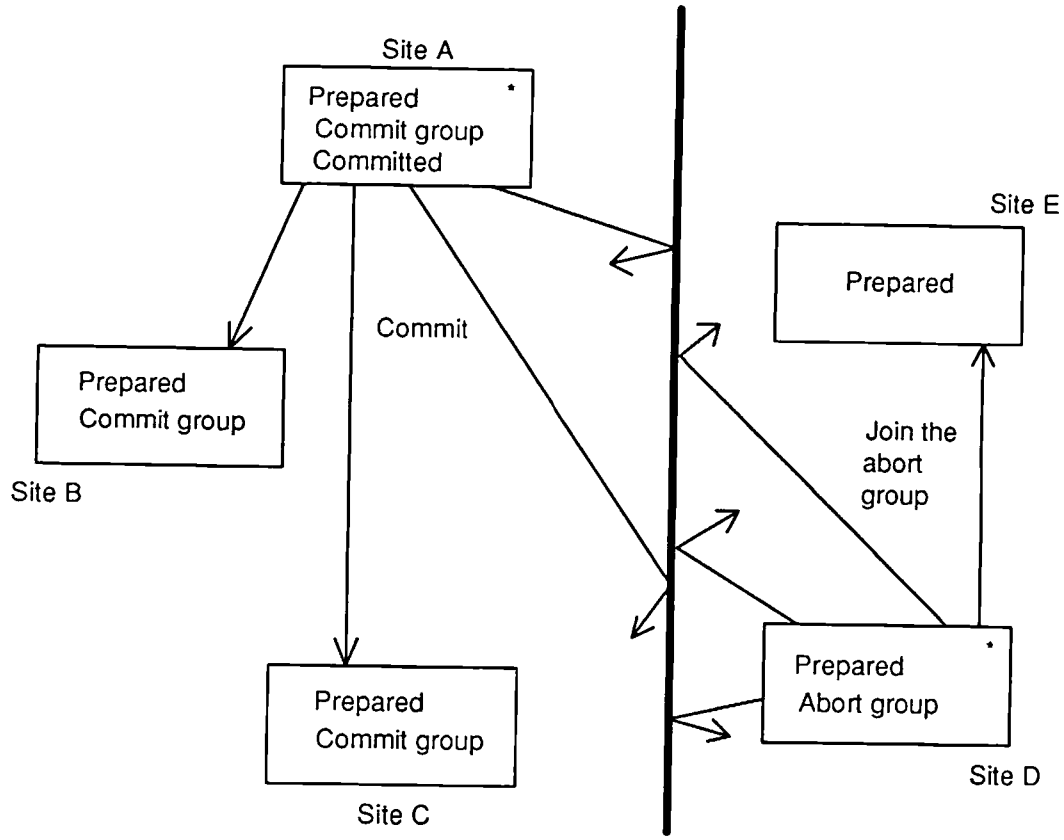

Figure I-4: Partition Example: Frame 4

The original coordinator commits because it succeeded in having a majority join the commit group. Based on its limited knowledge, the second coordinator has concluded (falsely) that not all sites are prepared, and so has joined the abort group. It is attempting (in vain) to get a majority of sites to join with it.

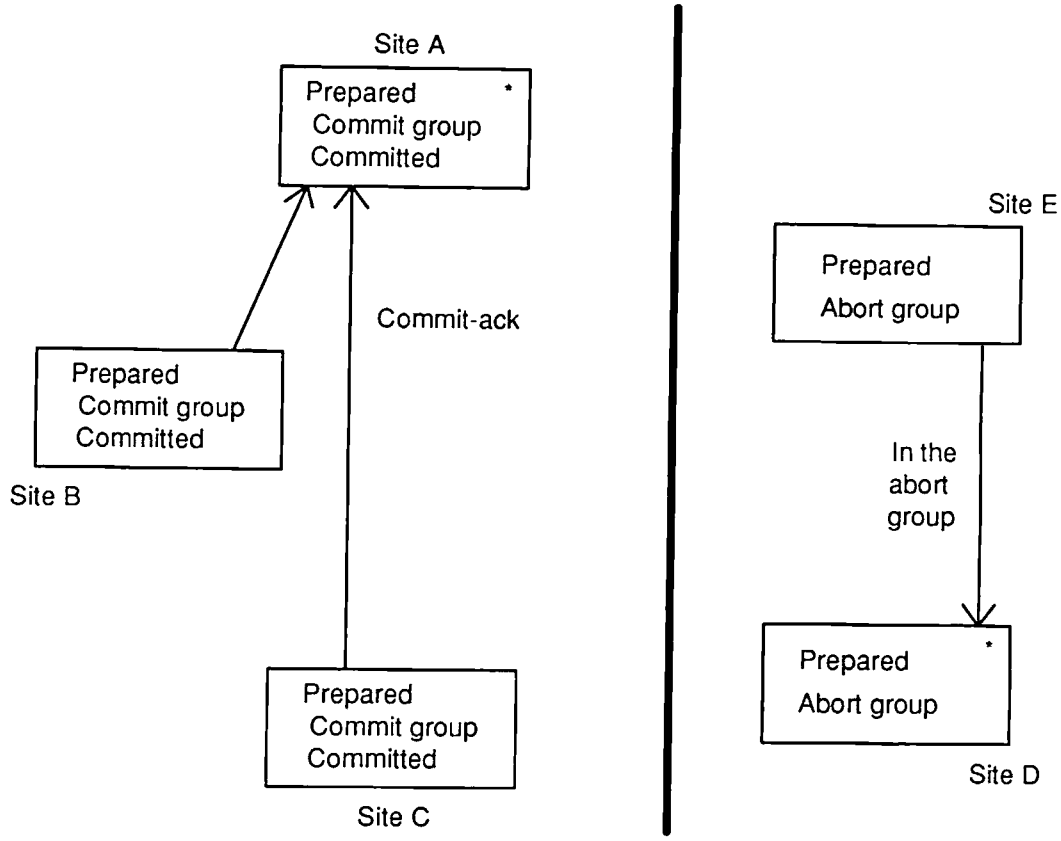

Figure I-5: Partition Example: Frame 5

Site E joins the abort group. Now every site has joined some group. The sites in the majority partition have committed, while the sites in the minority partition are doomed to block until the failure is repaired.

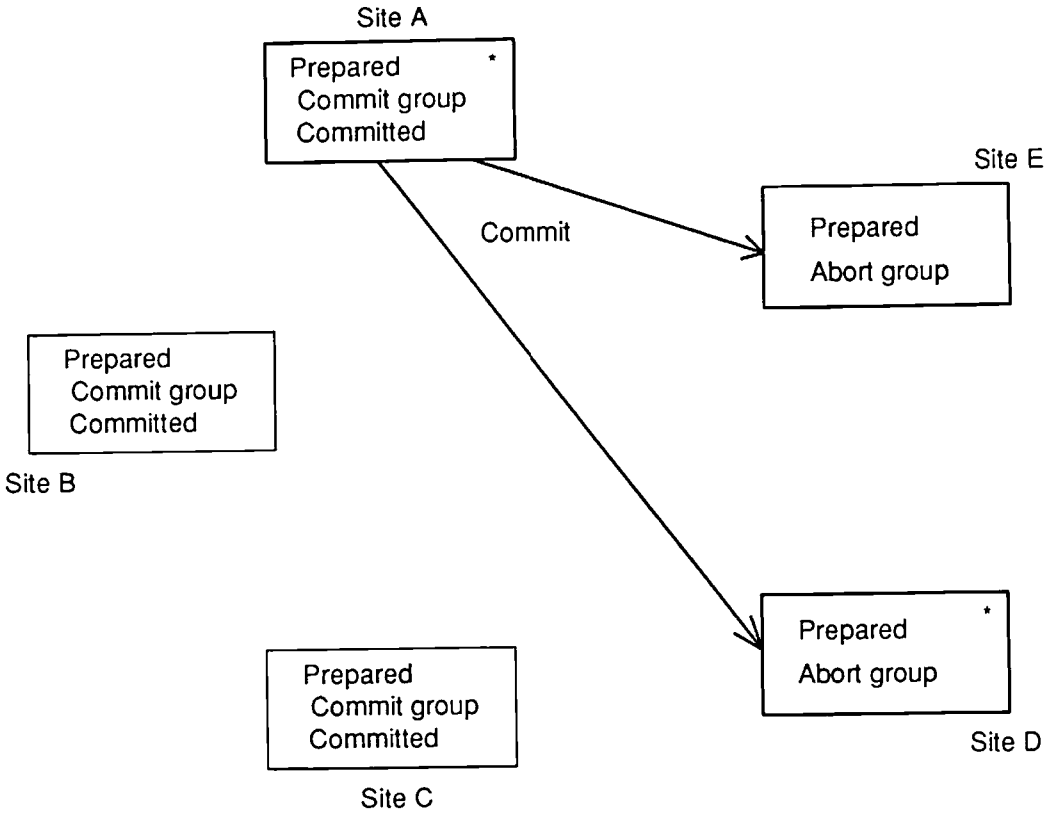

Figure I-6: Partition Example: Frame 6

The failure is repaired. The two coordinators are now dueling. Site A will continue sending "commit" until all sites acknowledge. Site D will continue sending "join-abort-group." (These messages are not shown.)

A subordinate receiving a join-abort-group message from D will respond with an "in-group" message that indicates its view of the state of every site. Therefore, the response from site E will contain no new information, while the responses from site B and C will list sites A, B, and C as committed. Likewise, site A will respond with an "outcome" message indicating commit. So the responses to D's messages have the same effect as A's messages: notice of commitment propagates. Receiving any type of information that indicates that any site has committed or aborted must result in the receiving site immediately committing or aborting, no matter what its previous state. Only one set of messages is shown to prevent clutter in the drawing.

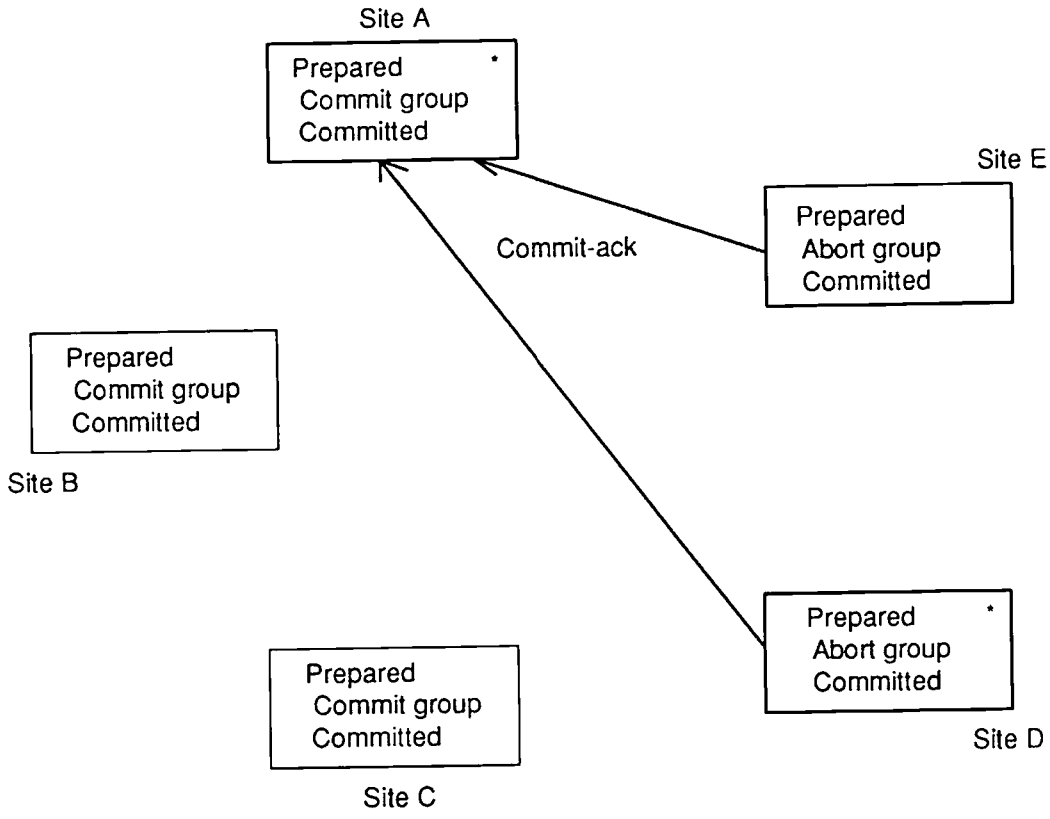

Figure I-7: Partition Example: Frame 7

Sites D and E commit, as instructed by coordinator A. D remains a coordinator.

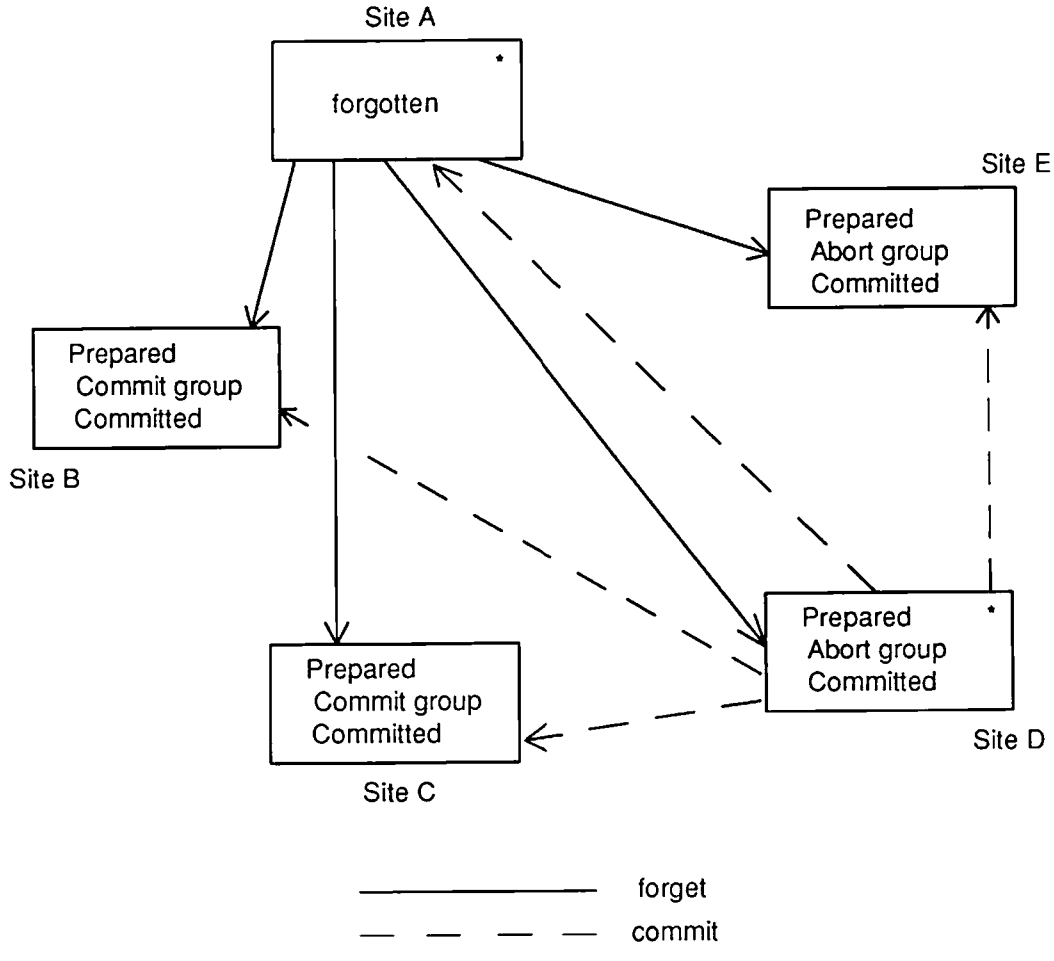

Figure I-8: Partition Example: Frame 8

The original coordinator has received commit acknowledgements from all sites. It now sends forget messages and then itself forgets.

Site D continues acting as a coordinator in the new state (committed) that it was pushed into: it sends commit messages.

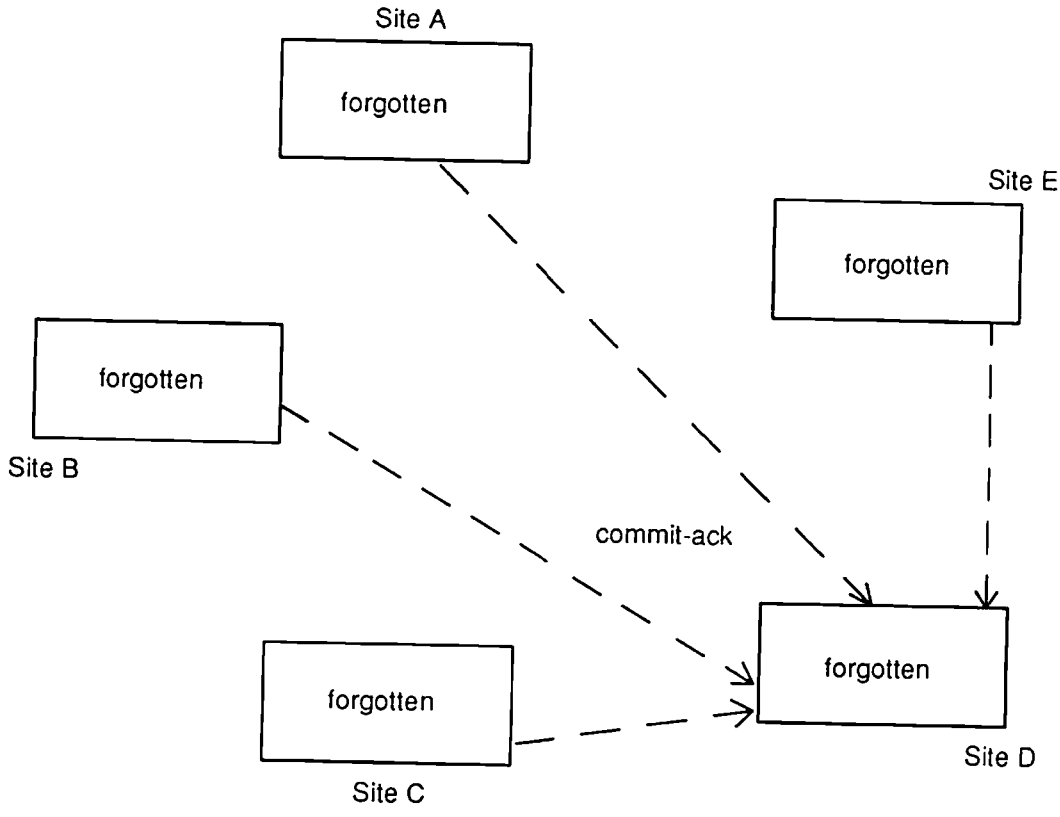

Figure I-9: Partition Example: Frame 9

For sites that receive two conflicting messages (subordinates B, C, and E), the order in which the messages (commit and forget) arrive does not matter. In either case — forgotten or not — the site will send commit-ack in response to the commit message. Site A, which has now forgotten about the transaction will also send commit-ack. In other words, the duel between the two coordinators will not prevent one from forgetting.

Site D then sends the forget message and forgets (this is not shown).

II. Complete Specification

This appendix contains a complete specification of the non-blocking commit protocol. The specification is sizable, and has been divided as follows: five sections (Sections II.1 through II.5) list what a site should do when:

1. it is a subordinate and receives an input (i.e., user-given "commit transaction" call or an inter-site message),
2. it is a coordinator and receives an input,
3. it has no information for the transaction and receives an input,
4. it is a subordinate and has just timed out waiting for a message,
5. it is recovering from a failure and is not yet accepting inputs.

Within each section, a separate table is devoted to every state. That is, one table lists what action to take in response to every type of input assuming that the site is in a given state. To shorten the specification, the commit and abort states are lumped into a single "terminated" state; likewise, the commit and abort messages are represented by a single (parameterized) "outcome" message.

Every action is justified in the text accompanying the table. An erroneous input is indicated by "..." An input causing a state transition is indicated by italics. The input which corresponds to expected normal operation is indicated by italic boldface. The expression "to broadcast" means to send a message to all sites not yet in the state commanded by the broadcast. The prepare, join-group, outcome, and forget messages are referred to as "commands." The other messages are called "acknowledgements."

II.1. Subordinate Actions

The specification for the subordinate has 35 actions (5 states times 7 messages). The next 5 tables specify action on a per-state basis. In every state, receipt of an acknowledgement message (prepare response, in-group, or outcome-ack) signals an error, since by definition a subordinate cannot have sent the corresponding command.

MESSAGE	ACTION
Prepare	***1. Collect votes of local processes.*** ***2. If some votes are no, vote no and begin aborting.*** ***3. If all votes are read-only, change state to read-only then send vote.*** ***4. If all votes are yes or read-only and some are yes, force prepare record, change state to prepared, then send vote.***
Prepare response	...
Join group	*If any site is in the commit group, error; otherwise, force in-group(abort) record, change state to in-group(abort), send in-group.*
In group	...
Outcome	*Commit is error. For abort, do as instructed: abort and drop locks, spool outcome record, and send outcome-ack.*
Outcome ack	...
Forget	...

Table II-1: Active Subordinate

Prepare is the normal case. Join-group must be for the abort group, since a transaction cannot commit unless every site is prepared or read-only. Likewise, outcome must indicate abort.

MESSAGE	ACTION
Prepare	Vote yes.
Prepare response	...
Join group	***Join winning group: force in-group record, send in-group.***
In group	...
Outcome	*Do as instructed: either abort and drop locks or just drop locks, spool outcome record. When record is forced, send outcome-ack.*
Outcome ack	...
Forget	...

Table II-2: Prepared Subordinate

A prepare message is a duplicate. Join-group is the normal case. Forget is an error because every site must be read-only or terminated before forget can be sent.

MESSAGE	ACTION
Prepare	Vote read-only.
Prepare response	...
Join group	*Join winning group: force in-group record, send in-group.*
In group	...
Outcome	...
Outcome ack	...
Forget	***Forget.***

Table II-3: Read-Only Subordinate

Same as prepared except that outcome is an error and forget is legal. A read-only site is never involved in the notify phase.

MESSAGE	ACTION
Prepare	Send prepare response. Since the prepare response lists the state of every known site, the message will indicate that this site is in a group.
Prepare response	...
Join group	Send in-group.
In group	...
Outcome	***Do as instructed: either abort and drop locks or just drop locks, spool outcome record, when forced send outcome-ack.***
Outcome ack	...
Forget	...

Table II-4: In-Group Subordinate

A prepare or join-group message is a duplicate. Outcome is the normal case. Forget is an error because every site must be read-only or terminated.

MESSAGE	ACTION
Prepare	Send prepare response. Since the prepare response lists the state of every known site, the message will indicate that this site is terminated.
Prepare response	...
Join group	Send in-group. Since the in-group message lists the state of every known site, the message will indicate that this site is terminated.
In group	...
Outcome	If outcome in message is opposite, error; else send outcome-ack.
Outcome ack	...
Forget	***Forget.***

Table II-5: Terminated Subordinate

Every message except forget is a duplicate.

These tables exhibit an elegant symmetry:

- Acknowledgement messages are errors.
- Excepting the read-only state and the forget message, commands earlier than the expected one are ignored, and commands later than expected are heeded.

II.2. Coordinator Actions

The specification for coordinator has 36 actions (5 states times 7 messages, plus the commit call from the user). Coordinator specifications are much more complicated than those for subordinates because, in addition to receiving acknowledgements sent by subordinates (the normal case), another coordinator can send commands.

MESSAGE	ACTION
User commit call	*1. Get the list of sites from the communication manager.* *2. Have all local processes vote.* *3. If some votes are no, invoke the abort protocol (it is usually faster).* *4. If all votes are read-only, change the state to read-only, then broadcast prepare messages.* *5. If all votes are yes or read-only and some are yes, force a prepare record, change the state to prepared, and broadcast prepare messages.*
Prepare	...
Prepare response	...
Join group	...
In group	...
Outcome	...
Outcome ack	...
Forget	...

Table II-6: Active Coordinator

A site can be an *active* coordinator only if it is the original coordinator. In that case, no other site is prepared yet, and so there can be no other coordinator. So the original coordinator should not receive any input except the user's commit call.

To process the commit call, the coordinator tries to prepare itself, then tries to prepare the subordinates.

MESSAGE	ACTION
Prepare	Vote yes.
Prepare response	***If some site is listed as being in the commit group, then broadcast join-group(commit) messages. If some site is listed as being committed or aborted, force the appropriate outcome record and broadcast outcome messages. If the response is a no vote, force an in-group(abort) indicator and broadcast join-group(abort) messages. If all responses are in, and all are either read-only or yes, then broadcast join-group(commit) messages.***
Join group	*Do as instructed, join the winning group: force an in-group record, broadcast join-group messages, respond in-group to the sender.*
In group	...
Outcome	*Do as instructed: force an outcome record, broadcast outcome messages, respond outcome-ack to the sender.*
Outcome ack	...
Forget	...

Table II-7: Prepared Coordinator

The only legal messages are those from another coordinator (prepare, join group, outcome, but not forget) or a prepare response. A forget message would be an error because forget should be sent only after every site has committed or aborted. This site, being not read-only, should be terminated before receiving the forget message.

A negative prepare response causes transition to the in-group(abort) state, but if every site prepares (the normal case), the coordinator delays joining the commit group. This is the optimization of Section 2.5.4.

A prepare message from another coordinator is simply acknowledged; the two coordinators are dueling, and both are in the same state. If told by another coordinator to join a group or commit/abort, then the other coordinator is in a more advanced state, and its command should be heeded.

MESSAGE	ACTION
Prepare	Vote read-only.
Prepare response	***If some site is listed as being in the commit group, then broadcast join-group(commit) messages. If some site is listed as being committed or aborted, force the appropriate outcome record and broadcast outcome messages. If the response is a no vote, force an in-group(abort) indicator and broadcast join-group(abort) messages. If all responses are in, and all are either read-only or yes, then broadcast join-group (commit) messages.***
Join group	*Do as instructed, join the winning group: force an in-group record, broadcast join-group messages, respond in-group to the sender.*
In group	...
Outcome	Send outcome-ack.
Outcome ack	...
Forget	*Forget.*

Table II-8: Read-Only Coordinator

Same as when prepared, except that there is no need to write a commit/abort record if an outcome message is received and that a forget message is not an error.

MESSAGE	ACTION
Prepare	Send join-group (specifying commit or abort, depending group).
Prepare response	Ignore.
Join group	*If sending site is higher ranked, join winning group (i.e, force an in-group record) and send in-group message. If sending site is lower ranked, send join-group.*
In group	***If some site is listed as being committed or aborted, force the appropriate outcome record and broadcast outcome messages. If there is already a quorum or if one would be achieved by the coordinator joining, then spool the appropriate in-group record, force outcome record and broadcast outcome.***
Outcome	*Do as instructed: force outcome record, broadcast outcome message, respond outcome-ack to sender.*
Outcome ack	...
Forget	...

Table II-9: In-Group Coordinator

The legal messages are those from another coordinator (except forget), the expected response from a subordinate (in-group), and a delayed or duplicate response from a subordinate (prepare response).

A delayed prepare response from a subordinate should be ignored. An in-group message represents normal operation. If the transaction has achieved a quorum or would achieve one if the coordinator joins, then the coordinator joins the group and then commits or aborts.

A prepare message from another coordinator indicates that the two coordinators are dueling, and the sender is in a less advanced state; accordingly, respond with a join-group message. A join-group message indicates that the dueling coordinators are in the same state: break the tie by ranking in the list of sites. If the sender is higher ranked, it dominates and makes this site joins a group. Otherwise, the receiver dominates. If told by another coordinator to commit/abort, then the other coordinator is in a more advanced state, and its command should be heeded.

MESSAGE	ACTION
Prepare	Send outcome.
Prepare response	Ignore.
Join group	Send outcome.
In group	Ignore.
Outcome	If outcome in message is opposite, error; else send outcome-ack.
Outcome ack	***If all subordinates have responded, broadcast forget, spool a done record, and forget.***
Forget	*Forget.*

Table II-10: Terminated Coordinator

Every message is legal when a coordinator is terminated.

A prepare response or an in-group message are delayed duplicates, so they should be ignored. An outcome-ack is normal.

Prepare and join-group messages come from a less advanced coordinator, and so an outcome message is sent in response. An outcome message comes from a coordinator in the same state, and so an outcome-ack is sent. Forget comes from a more advanced coordinator, and should be heeded.

II.3. Stateless Actions

MESSAGE	ACTION
Prepare	Vote no. Either this is a delayed duplicate, or the receiving site was an active or read-only subordinate that has crashed and recovered; to be safe, must assume that it was active.
Prepare response	Ignore. This is a delayed duplicate.
Join group	*Join winning group: force in-group record and send in-group message. Either this is a delayed duplicate, or the receiving site was an active or read-only subordinate that has crashed and recovered.*
In group	Ignore. This is a delayed duplicate.
Outcome	Send outcome ack. Either this is a delayed duplicate, or the coordinator needs your acknowledgement in order to forget. (you may have crashed before acking or the message may have been lost).
Outcome ack	Ignore. This is a delayed duplicate.
Forget	Ignore. This is a delayed duplicate.

Table II-11: Coping with a Message for an Unknown Transaction

Notice that — for messages that both protocols have in common — the behavior is identical to that of the Presumed Abort variation of two-phase commitment.

II.4. Timeout

Become coordinator in the current state.

II.5. Recovery

Become coordinator in the current state.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young.
Mach: A New Kernel Foundation for UNIX Development.
In *Proc. of Summer Usenix*, pages 93-112. July, 1986.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman.
Concurrency Control and Recovery in Database Systems.
Addison Wesley, 1987.

[3] D. Cheung and T. Kameda.
Site Optimal Termination Protocols for a Distributed Database Under Network Partitioning.
In *Proc. Fourth Ann. Symp. on Principles of Distributed Computing*, pages 111-121. ACM, August, 1985.

[4] D. Duchamp.
Protocols for Distributed and Nested Transactions.
In *Proc. Unix Transaction Processing Wkshp.*, pages 45-53. May, 1989.

[5] D. Duchamp.
Analysis of Transaction Management Performance.
In *Proc. Twelfth Symp. on Operating System Principles.* ACM, December, 1989.
to appear.

[6] C. Dwork and D. Skeen.
The Inherent Cost of Nonblocking Commit.
In *Proc. 2nd Ann. Symp. on Principles of Distributed Computing*, pages 1-11. ACM, August, 1983.

[7] H. Garcia-Molina.
Elections in a Distributed Computing System.
IEEE Trans. on Computers C-31(1):48-59, January, 1982.

[8] D. K. Gifford.
Weighted Voting for Replicated Data.
In *Proc. of the Seventh Symp. on Operating System Principles*, pages 150-162. ACM, December, 1979.

[9] J. N. Gray.
Notes on Database Operating Systems.
In R. Bayer, R. M. Graham, G. Seegmuller (editors), *Lecture Notes in Computer Science.* Volume 60: *Operating Systems - An Advanced Course*, pages 393-481. Springer-Verlag, 1978.

[10] M. Hammer and D. Shipman.
Reliability Mechanisms for SDD-1.
ACM Trans. on Database Systems 5(4):431-466, December, 1980.

[11] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan.
Recovery Management in Quicksilver.
ACM Trans. on Computer Systems 6(1):82-108, February, 1988.

[12] *Systems Network Architecture. Format and Protocols Reference Manual: Architecture Logic for LU Type 6.2*
SC30-3269-3 edition, IBM Corporation, 1985.

[13] T. V. Lakshman and A. K. Agrawala.
O(N$\sqrt{N}$) Decentralized Commit Protocols.
In *Proc. 5th IEEE Symp. on Reliability in Distributed Software and Database Systems*, pages 104-110. 1986.

[14] T. V. Lakshman and A. K. Agrawala.
Communication Structure of Decentralized Commit Protocols.
In *Sixth Intl. Conf. on Distributed Computing Systems*, pages 100-107. May, 1986.

[15] T. V. Lakshman and A. K. Agrawala.
Efficient Decentralized Consensus Protocols.
IEEE Trans. on Software Engineering SE-12(5):600-607, May, 1986.

[16] L. Lamport, R. Shostak, M. Pease.
The Byzantine Generals Problem.
ACM Trans. on Programming Languages and Systems 4(3):382-401, July, 1982.

[17] G. Le Lann.
A Distributed System for Real-Time Transaction Processing.
Computer 14(2):43-48, February, 1981.

[18] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, W. Weihl.
Argus Reference Manual.
Technical Report MIT/LCS/TR-400, MIT, November, 1987.

[19] C. Mohan, B. Lindsay, R. Obermarck.
Transaction Management in the R* Distributed Data Base Management System.
ACM Trans. on Database Systems 11(4):378-396, December, 1986.

[20] D. P. Reed.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, Massachusetts Institute of Technology, September, 1978.

[21] D. Skeen and M. Stonebraker.
A Formal Model of Crash Recovery in a Distributed System.
In *Proc. Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 129-142. 1981.

[22] D. Skeen.
Nonblocking Commit Protocols.
In *SIGMOD '81*, pages 133-142. 1981.

[23] D. Skeen.
Crash Recovery in a Distributed Database System.
PhD thesis, Univ. of California, Berkeley, May, 1982.

[24] D. Skeen.
A Quorum-Based Commit Protocol.
In *Proc. Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69-80. February, 1982.

[25] A. Z. Spector, R. Pausch, G. Bruell.
Camelot, A Flexible, Distributed Transaction Processing System.
In *Thirty-third IEEE Computer Society Intl. Conf. (COMPCON)*, pages 432-437. March, 1988.

[26] S. Yuan and A. K. Agrawala.
A Class of Optimal Decentralized Commit Protocols.
In *Eighth Intl. Conf. on Distributed Computing Systems*, pages 234-241. June, 1988.

[27] S. Yuan and P. Jalote.
Fault Tolerant Commit Protocols.
In *Proc. 5th Intl. Conf. on Data Engineering*, pages 280-286. 1989.