A Decentralized Termination Protocol*

Dana Sken

Computer Science Division
EECS Department
University of California
Berkeley, California

Abstract

The smallest recoverable unit of work in a distributed database system is a transaction. Whenever site failures leave the processing of a distributed transaction in a (potentially) unsafe state, a termination protocol is invoked to restore the database to a safe state enabling operational sites to proceed with future transactions. In this paper we propose one such termination protocol and sketch a proof of its correctness. The protocol is an example of a decentralized protocol, where each site assumes an equal and symmetric role. The proposed protocol is resilient to all combinations of site failures that do not partition the network.

1. Introduction

The smallest recoverable unit of work in a distributed database system is a transaction. Whenever site failures leave the processing of a distributed transaction in a (potentially) unsafe state, a termination protocol is invoked. The goal of a termination protocol is to move the database to a consistent state by either backing out the transaction at all participating sites or by (recoverably) installing the updates at all operational sites.

In this paper we propose one such termination protocol and sketch a proof of its correctness. The protocol is an example of a decentralized protocol. In a decentralized protocol, each site assumes an equal and symmetric role. This can be contrasted with the more popular centralized protocols where master/slave relationships exist among the sites.

The remainder of the paper is organized as follows. In the second section, we develop the necessary background material which includes defining and discussing termination and decentralized protocols. In the third section we introduce an example of a nonblocking commit protocol. The example is included for two reasons. First, an understanding of commit protocol is essential toward an understanding of termination protocols. Secondly, it is a good example of a decentralized protocol. In the fourth section, we present a decentralized termination protocol which is resilient to arbitrary site failures.

Throughout the paper, we assume that the underlying communications network provides point-to-point communications between any two operational sites (however, we do not require that messages be received in the order sent). We also assume that the network can detect and verify site failures by timeouts and by observing unsuccessful attempts at retransmission.

In addition to site failures, the proposed protocol can be made resilient to network partitions, arbitrary message loss, and to uncertainty in the type of failure observed. These extensions are outside the scope of this paper, but are discussed in [SKE68i].

2. Background

By definition a transaction on a distributed database system is a logically atomic operation: it must be processed at all sites or at none of them. Designing protocols for transaction management that are resilient to various failures, including arbitrary site failures and partitioning of the communications network, is a very difficult problem. We now discuss some of the aspects of resilient transaction management.

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, the transaction is immediately upon recovery the site will abort the transaction. Both commit and abort are irreversible.

The problem of guaranteeing transaction atomicity is compounded when more than one site is involved. Assuming that each site has a local recovery strategy which provides atomicity at the local level, the problem becomes one of insuring that the sites either unambiguously abort or unambiguously commit. A mixed decision results in an inconsistent database. Protocols for preserving transaction atomicity are called commit protocols. Several commit protocols have been proposed ([ALSB78, ELL77, GRAY79, HAMM79, LAMP78, LIND78, SKE68i, STON78]).

For many applications it is intolerable for operational sites to be forced to indefinitely block the progress of a transaction until a failed site has recovered. Instead, it is preferable for the operational site to abort the transaction (if necessary) so that the locks required by the transaction can be released. Commit protocols that never leave transaction processing in a state where operational sites must wait until the recovery of a failed site before a consistent commit decision (e.g., abort or commit) can be reached are called nonblocking. Non-blocking protocols have been proposed in [HAMM79, [SKE68i].
The Protocol

The nonblocking protocol is derived from the simple protocol by adding another message round and delaying the commit point of a transaction until the end of the second round.

In the simple commit protocol, a site would commit at the end of the single message round if all sites had voted yes. In the nonblocking version of the protocol, an all yes vote would trigger a second round of messages, where each site sends prepared to commit messages and waits. Upon receiving prepared to commit messages from all its cohorts, a site will then commit the transaction. (The protocol is given in its entirety in Figure 1.)

Whenever a site detects the failure of another site while executing the commit protocol, it will invoke a termination protocol. The detection of the failure and the subsequent invocation can occur during either message round.

Properties of Nonblocking Commit Protocols

In the nonblocking decentralized commit protocol, we can identify five distinct states in processing a transaction: the state where the site is waiting to receive the transaction; the all yes vote state where the site has not yet received all yes votes; the prepared state where the site has sent prepared to commit messages and is waiting for a similar message from all cohorts and two final states, abort and commit.

The transaction states of any commit protocol can be partitioned into two sets: commitable and noncommitable. A state is called commitable if occupancy of that state by any site implies that all sites have voted yes on committing the transaction. A state that is not commitable is called noncommitable. In the nonblocking commit protocol presented above, both the initial and second round are commitable states.

Initial Phase. Transaction is sent to all sites.

First Round. Each site broadcasts its vote, yes or no, for the transaction.

- If a site receives all yes votes during this round, then a second round is initiated. Otherwise, the site aborts the transaction.

Second Round. Each site broadcasts a prepared to commit message.

- Upon receiving a prepared... message from each of its cohorts, a site commits the transaction.

(To aid noncommitable states in practice, one might consider a practical solution for commitable states since a transaction that is not in a final commit state at any site can still be aborted.)
prepare state and the commit state are committable states; the remaining states are noncommittable.

All nonblocking protocols exhibit the following properties (see [SKES91b]):

(1) all operational sites occupy committable states before the transaction is committed at any site.
(2) all operational sites occupy noncommittable states before the transaction is aborted at any site.

4. A Decentralized Termination Protocol

A termination protocol must guarantee that every operational site terminates the transaction in a consistent state. The correct execution of a termination protocol depends on the properties of commit protocols described in the previous section.

Two issues complicate the design of a termination protocol. First, it must be resilient to subsequent site failures. And second, sites may detect a given failure at different points in their protocol. For example, some sites may detect a failure in round one of the nonblocking decentralized protocol while others will not detect it until the second round.

First, we will present a simple decentralized termination protocol that is not resilient to further site failures during its execution. This will serve to introduce the basic ideas used in a decentralized termination protocol.

We will then present an extension of the simple protocol that is resilient to all combinations of site failures that do not partition the network. Normally the resilient termination protocol will require two rounds of message interchanges; however, additional site failures during the execution of the protocol may cause additional rounds. The maximum number of rounds is equal to the initial number of operational sites.

To simplify notation, we will speak as though sites sent messages to themselves during a round. We will also refer to operational sites simply as the participants.

A Simple Termination Protocol

The protocol consists of a single round of messages. During this round, the message sent by a site is determined solely by its current transaction state. There are three possible messages:

- abort if the transaction state is a final abort state.
- committable if the transaction state is a committable state, and
- noncommittable if the transaction state is neither a committable state nor an abort state.

Upon receiving messages from all the participants, a site will move directly to a final state according to the following rule:

Simple Commit Rule. If at least one committable message is received, then commit the transaction; otherwise, abort it.

As an example of using the protocol, consider invoking it from the nonblocking decentralized commit protocol described in Section 3. A site will send an abort message if it currently occupies the abort state; it will send a committable message if it currently occupies the prepared state or the commit state; and it will send a noncommittable message if it occupies either the initial state or the wait state.

It is straightforward to argue the correctness of the protocol. We observe that the transaction is committed if and only if one of the participants is initially in a committable state. From the properties of nonblocking commit protocols given in Section 3, we know that occupancy of a committable state at any site implies that all sites can commit the transaction. Furthermore, it implies that no site has aborted the transaction. Therefore, we conclude that the simple termination protocol is correct.

This protocol is not very robust as is demonstrated in the following scenario involving three sites. Let Site 1 be the only site in a committable state at the termination protocol, and let Site 1 fail after sending a committable message to Site 2. At the end of the first message round, Site 2 would have received one committable message (from Site 1) and one noncommittable message (from Site 3). Site 3 would have received no messages from Site 1 and a noncommittable message from Site 2. Clearly, Site 3 cannot safely proceed until it queries Site 2 as to the state of the failed site. If Site 2 fails at this point, then Site 3 must block the transaction.

The protocol cannot be made more robust by changing the commit rule. For example, if the rule was to commit only after all sites had sent committable messages, then a blocking scenario that is the mirror image of the above scenario could be contrived. It is fairly intuitive (and can be shown formally) that no "single round" termination protocol is resilient to arbitrary site failures.

A Resilient Termination Protocol

The design of a resilient "multiple-round" termination protocol is complicated by two subtle issues. The first issue is that an operational site may fail immediately after making a commit decision (and therefore be unavailable to participate in subsequent message rounds). This was the case in our previous scenario where Site 2 failed after committing the transaction. The second issue is that often a given site does not know the current operational status (i.e., "up" or "down") of the other sites. In particular, upon entry into a termination protocol, the identities of the other operational sites may not be known.

The second issue can lead to very subtle problems. Again consider the scenario where Site 1 sends a committable message to Site 2 and then crashes. Site 2 sends out noncommittable messages, receives the committable message from Site 1, commits, and then promptly fails. Now, Site 3 receives a single noncommittable message (from Site 2). Let us assume that Site 3 was not aware that Site 1 was up at the beginning of the protocol (a reasonable assumption). Then, Site 3 would not suspect that the messages it received were inconsistent with those received by Site 2, and it would make an inconsistent commit decision.2

We have argued that a resilient protocol requires at least two rounds. The protocol that we now present requires exactly two message rounds when no site failures occur during its execution. Unfortunately, in the worst case, each site failure may require an additional message round.

---

2This illustrates that single round protocols sometimes make incorrect decisions when both additional site failures occur and the information concerning the status of operational sites is incomplete. Furthermore, the inconsistent decisions go undetected unless additional message rounds are added.
The protocol presented is an extension of the simple protocol. The same three messages—abort, committable, and noncommittable—will be used again in the first round and in all subsequent rounds.

The sending of messages during the first round proceeds as before: a site examines its transaction state and sends the appropriate message. However, the actions triggered by the receipt of the messages differ from before.

To define the remainder of the protocol we must specify:
1. the rules for the messages sent during the subsequent rounds,
2. the rules for moving to a final transaction state (i.e., either commit or abort), and
3. the rules for terminating the protocol (this is closely linked to 2).

These rules are obviously interrelated, but we will treat them sequentially.

The rules for sending messages are simpler and will be discussed first. The messages sent by a site in the second round and subsequent rounds will be determined solely by the messages received during the previous round. The receiver is reminded that during a round a site sends the same message to all (operational) participants, including itself. This message to itself, as any other message, will be used in determining the next round of messages.

There are three cases which are treated in the next three paragraphs. The rules for sending messages are summarized in Figure 2a.

The receipt of an abort message by a site during any round implies that the sender has aborted the transaction. Therefore, in subsequent rounds the site will send abort messages.

The receipt of a single committable message during the first round implies that the transaction was committable at the sender, and therefore, it is committable at all sites. The receiver of the committable message, being informed that the transaction is committable, should send committable messages during all subsequent rounds. Similarly, a committable message received during a subsequent round implies that all sites can commit, and will trigger the sending of committable messages in all of the later rounds.

If only noncommittable messages are received during a round, then the site must send noncommittable messages in the next round. From the above three rules, we infer:

**Lemma 1.** Once a site begins sending a committable (abort) message, it will send that message in all subsequent rounds.

We now turn our attention to rules for committing and aborting the transaction. Clearly, if a site ever receives an abort message, it should immediately abort the transaction because the transaction has been aborted at other sites (in particular, it was aborted by the sender of the message). However, committing a transaction is not so straightforward.

Recall that a major flaw with the simple termination protocol is that a site commits after receiving a single committable message. We require a rule analogous to property (1) of nonblocking commit protocols, which

---

3

**States that all sites must be in a committable state before any site commits. This leads us to the following rule:**

**Commit Rule.** A transaction is committed at a site only after the receipt of a round comprising entirely of committable messages.

Before continuing with the termination rules for the protocol, it will be instructive to look at a "worst case" execution of the protocol. The execution is worst case in the sense that the maximum number of message rounds is required before the transaction is committed. Only the rules previously discussed are used.

The worst case execution for five participants is illustrated in Figure 3. (In the figure the messages received by a site during a round comprise a vector, where the \( i \)th component is the message received from the \( i \)th site. \( C_A \) and \( N \) are abbreviations for committable, abort, and noncommittable. A dash (--) indicates that no message was received from that site.)

Initially, the first site is the only one in a committable state. It fails after sending a single message that is addressed to the second site. In general, during the \( k \)th round, the \( k \)th site fails after sending a single committable message (to the \( k+1 \)th site). Therefore, during each round one more site becomes aware that the transaction is committable. This continues until the fifth round, where Site 5 is the sole remaining operational site and it commits the transaction.
MESSAGES RECEIVED

<table>
<thead>
<tr>
<th>SITE 1</th>
<th>SITE 2</th>
<th>SITE 3</th>
<th>SITE 4</th>
<th>SITE 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial state</td>
<td>commitable</td>
<td>non</td>
<td>non</td>
<td>non</td>
</tr>
<tr>
<td>round 1</td>
<td>(1)</td>
<td>C N H N N</td>
<td>- N N N N</td>
<td>- N N N</td>
</tr>
<tr>
<td>round 2</td>
<td>FAILED</td>
<td>(1)</td>
<td>C N N N</td>
<td>- - N N N</td>
</tr>
<tr>
<td>round 3</td>
<td>FAILED</td>
<td>FAILED</td>
<td>(1)</td>
<td>- - C N N</td>
</tr>
<tr>
<td>round 4</td>
<td>FAILED</td>
<td>FAILED</td>
<td>FAILED</td>
<td>(1)</td>
</tr>
<tr>
<td>round 5</td>
<td>FAILED</td>
<td>FAILED</td>
<td>FAILED</td>
<td>FAILED</td>
</tr>
</tbody>
</table>

NOTE: (1) site fails after sending a single message.

Figure 3. Worst case execution of the resilient termination protocol.

Now let us consider the problem of correctly terminating the protocol. If a site eventually receives at least one abort message or eventually receives commitable messages from all sites, then there is no problem. However, it is possible for the transaction to progress to a state where all sites are sending noncommitable messages. The protocol must be able to detect this situation and abort the transaction. We will use the following rule to terminate such transactions.

**Termination Rule.** If a site ever receives two successive rounds of noncommitable messages and it detects no site failures between the rounds, then it can safely abort the transaction.

We will justify this rule later.

We can make one final enhancement to the protocol. Notice that all sites may not decide to abort at the same time. For example, let there initially be only one site in an abort state, and let the remaining participants be in a noncommitable state. If the site in the abort state fails while sending messages in the first round, then those participants receiving an abort message will immediately abort the transaction, while the others will continue with subsequent message rounds. To expedite the abortion of the transaction at all sites, those sites aborting the transaction during the first message round should participate in later rounds. Therefore, we will always require a site to participate in one additional message round after aborting the transaction. Note that this is only a "performance" enhancement; the protocol will eventually abort the transaction at all sites irrespective of whether the sites aborting the transaction participate in the additional message round.

The commit and termination rules are summarized in Figure 2b.

**Correctness Argument**

To demonstrate correctness we must show (1) that the protocol always terminates, and (2) that it terminates in a consistent state. We will show termination first.

Let n be the number of participants at the beginning of the protocol. Let \( N(r) \) be the set of sites sending noncommitable messages to site i during round r.

We have:

**Lemma 2.** \( N(r+1) \subseteq N(r) \)

**Proof.** This follows directly from Lemma 1: for a site to send a noncommitable message in round \( r+1 \), it must have sent a noncommitable message in round r.

**Lemma 3.** If \( N(r+1) = N(r) \neq \emptyset \), then all messages received by site i during both rounds r and \( r+1 \) were noncommitable messages.

**Proof.** Without loss of generality assume that site i is operational. The argument proceeds by contradiction. Let \( N(r+1) = N(r) \neq \emptyset \) and let round r contain messages other than noncommitable messages. We will only discuss the case where commitable messages appear. There are two subcases depending on the message sent by i during round r:

**Case 1.** Site i sends a noncommitable message during round r. In round r+1, it will send a commitable message because it received a commitable message during round r (by assumption). This contradicts the claim \( N(r+1) = N(r) \).

**Case 2.** Site i sends a commitable message during round r. Since site i did not fail in round r, all sites received a commitable message from i during that round. Therefore, in round r+1 all sites will send commitable messages. Again this is a contradiction.

Lemmas 2 and 3 show that the number of sites sending noncommitable messages either monotonically decreases toward zero with each round, or two rounds will occur with the same number. In the former case, the transaction will be terminated by the time the number reaches zero (and this requires at most n rounds). In the latter case, the transaction will be aborted because of the termination rule.

To show that a consistent state is reached, we require the following results:
Lemma 4. During any message round, abort and committable messages may not both be sent.

Proof. The proof for the first round follows directly from the properties of nonblocking commit protocols; it is never the case that one site is in an abort state while another site is in a committable state.

From the rules for sending messages, we know that a round can include a certain type of message only if that message type was present in the previous round. (This follows from the observation that a given message type must be received by a site before it will be sent by a site in the next round.) By induction, a message type can appear in a later round only if it was present in the first round. This observation proves the lemma.

Lemma 4 proves that it is never the case that some sites are trying to abort the transaction by sending abort messages, while others are trying to commit the transaction by sending committable messages. The commit rule ensures that sites begin to commit only after all operation sites are aware that the transaction is committable. Finally, the properties of a nonblocking commit protocol ensure that no site has aborted the transaction after a single site has entered a committable state. Collectively, these results imply the correctness of the protocol.

5. Conclusions

We have presented a termination protocol that is resilient to arbitrary site failures that do not partition the network. In [SKEE81a] this protocol is extended to handle network partitions.

The proposed termination protocol is an example of a decentralized protocol. These protocols have several advantages over centralized protocols—namely, they tend to be much simpler and easier to implement. Both of these advantages are derived from the symmetry inherent in all decentralized protocols.

The major disadvantage of decentralized protocols is the number of messages exchanged during a round (the number of messages is quadratic in the number of participants). In network environments where either control messages are cheap or a broadcast facility is available or both (e.g., an ETHERNET), the message cost is reasonable. Moreover, in realistic environments a site failure should be a rare event; therefore, the cost of the termination protocol should not be a significant issue.

Since message rounds are costly, an important design goal for any decentralized protocol is to minimize the number of rounds. It is easy to show that any resilient protocol requires a minimum of two message rounds before it can commit a transaction and, in the worst case, requires an additional message round for each failure detected ([SKEE81c]). The proposed protocol meets these lower bounds. In particular, it requires exactly two rounds when no additional site failures occur during its execution. Furthermore, a worst-case execution of the protocol is extremely rare in practice.

Finally, the proposed protocol is an optimistic protocol—it will commit the transaction whenever it is safe to do so—and it can be used in conjunction with any nonblocking commit protocol. In environments where messages are expensive, it is reasonable to run a centralized commit protocol and the proposed decentralized termination protocol.

REFERENCES