

Figure 10.6(b) Coordinator in three-phase commit.

should the coordinator send any commit message prior to sending the last of the prepare-commit messages, then blocking is possible.

What is essential about three-phase commit is that the coordinator sends all of the prepare-commit messages out before it sends any commit message. The intuitive reason is that the prepare-commit message informs each participant that all are willing to commit. If any participant T_i receives commit, it knows that the coordinator has sent all its prepare-commit messages, and thus every participant that is still live has received prepare-commit or is about to do so, since the message could be delayed but not lost by the network. That is, the receipt of a commit message by T_i tells T_i that all know all are willing to commit.

Technically, T_i only knows that every participant T either knows that all are willing to commit, or T will know it shortly, or T will fail before it receives the prepare-commit. However, since the protocol of Figure 10.6 only involves messages between the coordinator and participants, and because as-

sumption (5) assures us messages are not lost, it can be assumed that messages are received instantaneously. That is, when T_i commits, every participant has either received prepare-commit or has already failed. The reason is that if some T_j actually fails after the time T_i receives commit, but before T_j receives prepare-commit, then there would be no observable change in the activity of the network if we assumed that T_j had failed before T_i received commit. What we have shown is that it is impossible for two participants to be simultaneously in the Willing-to-commit and Committed states, respectively. This fact and other useful observations about the protocol of Figure 10.6 are summarized in the following lemma.

Lemma 10.1: Prior to transactions entering the recovery state, and under the (justifiable) assumption that messages are delivered instantaneously, the following states are incompatible.

- One (live or failed) participant cannot have entered the Committed state while any live participant is still in the Willing-to-commit state.
- One (live or failed) participant cannot have entered the Aborted state while another (live or failed) participant has entered the Committed state, or any live participant has entered the Ready-to-commit state.⁹

Proof: For (a), we note that in order for a participant to enter the Committed state before any recovery takes place, it must receive a commit message. By the argument given above, we know that every live participant has (on the assumption of instantaneous messages) received prepare-commit, and therefore has left the Willing-to-commit state.

We leave (b) as an exercise. The reader has only to examine Figure 10.6 and argue that a prepare-commit message cannot be sent if one or more participants have aborted. \square

Recovery in Three-Phase Commit

The consequence of Lemma 10.1 is that we cannot have a failed participant that has aborted if any live transaction has reached as far as the Ready-to-commit state, and we cannot have a failed participant that has committed if any live transaction is still in the Willing-to-commit state. Thus, when one or more participants detect the need for recovery, because of a timeout, we have only to arrange that each live participant discloses to the others its state, or more precisely, its state just before it entered the Recovery state. If all are in Willing-to-commit or Aborted, then we know no failed participant has committed, and it is safe for all to abort. If any has reached the Ready-to-commit state or the

⁹ In fact, it is not even possible for a failed participant to have entered Ready-to-commit, but we state the conditions this way because we want them to be weak enough that they are preserved during the recovery process.

Committed state, then no failed transaction can have aborted, so it is safe for all to commit.

In the latter case, the distributed commitment process must be taken by steps. That is, any participants still in the Willing-to-commit state must first be brought to the Ready-to-commit state, and then all those in that state must be made to commit. The reason we must continue in stages is that at any time, more participants may fail, and we must avoid creating a situation where one participant is in Willing-to-commit while another has already committed.

Electing a New Coordinator

As with two- or three-phase commit in general, the recovery process can be conducted in several different ways. As we have considered only the centralized, or coordinator-based approach, because it tends to save messages, let us continue with that approach now. Then as soon as one participant realizes recovery is needed, it sends a message to all the other participants. Several participants may reach this conclusion at about the same time, so many redundant messages will be sent in the worst case, but not in the typical case.

Then, the live participants must attempt to elect a new coordinator, because the only time we enter the Recovery state is if a participant has timed out waiting for the coordinator to send a message. Each participant knows the original set of participants, although some now are failed. We may assume that the participants are numbered T_1, \dots, T_k , and the lowest-indexed live participant will be the new coordinator. Since T_1 may have failed, we cannot just assume T_1 is the new coordinator. Rather, each participant must make known to the others that it is live. If done properly, at most one live participant will conclude that it is the new coordinator (because it never heard from any lower-numbered participant).

One relatively efficient way to make the decision is for each T_i to send a message with its index, i , to $T_{i+1}, T_{i+2}, \dots, T_k$ in that order. However, if T_i receives a message from a lower-numbered participant, then T_i knows it is not the coordinator, and so stops sending messages. Most participants will stop sending messages very quickly, but if some messages are delayed inordinately,¹⁰ then on the order of k^2 messages could be sent.

After this step, each live participant will have a notion of who the new coordinator is. If no failures occurred during the election, then all will have the same notion. However, if the lowest-numbered participant failed during the election, then there may be disagreement regarding who is the coordinator.

¹⁰ Note we are no longer assuming messages are sent instantaneously; that assumption was justified only by the pattern of messages (to and from the coordinator) that is present in the basic three-phase commit algorithm.

Example 10.6: Suppose there are participants T_1, \dots, T_4 . Also suppose that during the election, the following sequence of events occurs.

1. T_1 sends a message to T_2 before T_2 can send its own message to T_3 . Thus, T_2 never sends any messages.
2. T_1 fails.
3. T_3 sends a message to T_4 . T_4 is thereby inhibited from sending any messages.

The net effect of these events is that T_2 thinks T_1 is the coordinator, while T_3 and T_4 both think T_3 is the coordinator. After a suitable timeout period, so it can be determined that no more messages are being sent, T_3 starts its roll as coordinator by requesting the state of all participants.¹¹ \square

It is easy to show that no more than one live participant can think it is the new coordinator. For suppose T_i and T_j both are live and think they are the coordinator, where $i < j$. Since T_i thinks it is the coordinator, it never received a message from any participant lower than i . Thus, it continued to send out messages to the participants numbered above i , and in particular to T_j . Thus, T_j would not think it is the coordinator.

It is possible that no live participant thinks it is the coordinator, in which case the live participants will time out waiting for the recovery to begin. They will then elect a new coordinator.

The Recovery Algorithm

With these tools, we can describe an appropriate recovery algorithm to use with three-phase commit. This recovery strategy has the property that it never causes a participant to block, as long as at least one participant remains live. Unfortunately, it is not possible to avoid blocking in a situation where all participants fail, and then one recovers and finds it is in the Willing-to-commit state. As discussed in Example 10.4, in connection with two-phase commit, such a participant cannot rule out the possibility that some other participant is aborted, nor can it rule out the possibility that another committed. Thus, it must block and wait for more participants to recover. The steps taken for recovery are summarized as follows:

1. The live participants elect a new coordinator.
2. The new coordinator sends messages to all participants requesting their state immediately prior to recovery, which must be Aborted, Willing-to-commit, Ready-to-commit, or Committed. Failed participants, of course, will not reply, so the coordinator waits for the timeout period and then

¹¹ The timeout period need not be long. It can be based on the expected time for each T_i to receive a message from T_1 . If some T_i thinks it is the coordinator and isn't, it will get a message to that effect from some participant.

and there is nothing else we can do. However, if there are other copies of A , then we can proceed as if the copy at N did not exist. When N recovers, it not only has the responsibility to find out about the transactions being committed or aborted when it failed, but now it must find out which of its items are out of date, in the sense that transactions have run at the other sites and modified copies of items that, like A , are found at N and also at other nodes.

Obtaining Up-to-Date Values

When the failed site resumes activity, it must obtain the most recent values for all its items. We shall suggest two general strategies for doing so.

1. If site M discovers that site N has failed, M records this fact in its log. When N recovers, it sends a message to each site. If M receives such a message, M examines its log back to the point where it discovered N had failed, and sends the most recent value it has for all items it holds in common with N .¹⁵ The values of these items must be locked while the recovery of N is in progress, and we must be careful to obtain the most recent value among all of the sites with copies. We can tell the most recent values, because all transactions that have committed a value for item A must have done so in the same order at all the sites of A , provided we have a correct locking method. If we are using timestamp-based concurrency control, the write-times of the values determine their order.
2. All copies of all items may be assigned a write-time, whether or not timestamp concurrency control is in use. When a site N recovers, it sends for the write-times of all its items, as recorded in the other sites. These items are temporarily locked at the other sites, and the current values of items with a more recent write-time than the write-time at N are sent to N .

This description merely scratches the surface of the subject of crash management. For example, we must consider what happens when a site needed to restore values to a second site has itself failed, or if a site fails while another is recovering. The interested reader is encouraged to consult the bibliographic notes for analyses of the subject.

10.8 DISTRIBUTED DEADLOCKS

Recall from Section 9.1 that we have simple and elegant methods to prevent deadlock in single-processor systems. For example, we can require each transaction to request locks on items in lexicographic order of the items' names. Then it will not be possible that we have transaction T_1 waiting for item A_1 held by

¹⁵ Note that under the methods of locking and commitment described in this chapter, M must discover N has failed if there is a transaction that involves any item held by both N and M , so N will hear of all its out-of-date items.

T_2 , which is waiting for A_2 held by T_3 , and so on, while T_k is waiting for A_k held by T_1 . That follows because the fact that T_2 holds a lock on A_1 while it is waiting for A_2 tells us $A_1 < A_2$ in lexicographic order. Similarly, we may conclude $A_2 < A_3 \cdots A_k < A_1$, which implies a cycle in the lexicographic order, an impossibility.

With care, we can generalize this technique to work for distributed databases. If the locking method used is a centralized one, where individual items, rather than copies, are locked, then no modification is needed. If we use a locking method like the k -of- n schemes, which lock individual copies, we can still avoid deadlocks if we require all transactions to lock copies in a particular order:

1. If $A < B$ in lexicographic order, then a transaction T must lock all the copies of A that it needs before locking any copies of B .
2. The copies of each item A are ordered, and a transaction locks all copies of A that it needs in that order.

Even if it is possible under some circumstances to avoid deadlock by judicious ordering of copies, there is a reason to look elsewhere for a method of dealing with deadlocks. We discussed in Example 9.21 why it is sometimes difficult to predict in advance the set of items that a given transaction needs to lock. If so, then locking needed items in lexicographic order is either not possible or requires the unnecessary locking of items.

In the remainder of this section we shall take a brief look at some general methods for deadlock detection and deadlock avoidance that do not place constraints on the order in which a transaction can access items. First, we consider the use of timeouts to detect and resolve deadlocks. Next, the construction of a waits-for graph is considered as a detection mechanism. Finally, we consider a timestamp-based approach to avoiding deadlocks altogether.

Deadlock Resolution by Timeout

A simple approach to detecting deadlocks is to have a transaction time out and abort if it has waited sufficiently long for a lock that it is likely to be involved in a deadlock. The timeout period must be sufficiently short that deadlocked transactions do not hold locks too long, yet it must be sufficiently long that we do not often abort transactions that are not really deadlocked.

This method has a number of advantages. Unlike the waits-for-graph approach to be described next, it requires no extra message traffic. Unlike the timestamp-based methods to be described, it does not (usually) abort transactions that are not involved in a deadlock. It is prone, however, to aborting all or many of the transactions in a deadlock, rather than one transaction, which is generally sufficient to break the deadlock.

Waits-for-Graphs

We mentioned in Section 9.1 that a necessary and sufficient test for a deadlock in a single-processor system is to construct a *waits-for graph*, whose nodes are the transactions. The graph has an arc from T_1 to T_2 if T_1 is waiting for a lock on an item held by T_2 . Then there is a deadlock if and only if there is a cycle in this graph. In principle, the same technique works in a distributed environment. The trouble is that at each site we can maintain easily only a *local* waits-for graph, while cycles may appear only in the *global* waits-for graph, composed of the union of the local waits-for graphs.

Example 10.7: Suppose we have transactions T_1 and T_2 that wish to lock items A and B , located at nodes N_A and N_B , respectively. A and B may be copies of the same item or may be different items. Also suppose that at N_A , (a subtransaction of) T_2 has obtained a write-lock on A , and (a subtransaction of) T_1 is waiting for that lock. Symmetrically, at N_B T_1 has a lock on B , which T_2 is waiting for.

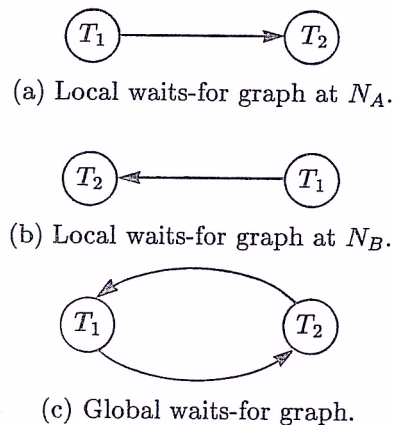


Figure 10.7 Global deadlock detection.

The local waits-for graphs at N_A and N_B are shown in Figure 10.7(a) and (b); clearly each is acyclic. However, the union of these graphs is the cycle shown in Figure 10.7(c). As far as we can tell at either of the sites N_A or N_B , there might not be a deadlock. For example, from N_A alone, we cannot be sure that anything prevents T_2 from eventually committing and releasing its lock on A , then allowing T_1 to get the lock. \square

Example 10.7 illustrates why in order to detect cycles it is necessary to send messages that allow a global waits-for graph to be constructed. There are

1. Use a central node to receive updates to the local waits-for graphs from all of the sites periodically. This technique has the advantages and disadvantages of centralized methods of locking: it is vulnerable to failure of the central node and to concentration of message traffic at that site,¹⁶ but the total amount of traffic generated is relatively low.
2. Pass the current local waits-for graphs among all of the sites, preferring to append the local graph to another message headed for another site if possible, but sending the local graph to each other site periodically anyway. The amount of traffic this method generates can be much larger than for the central-node method. However, if the cost of messages is relatively invariant to their length, and frequently waits-for information can be “piggybacked” on other messages, then the real cost of passing information is small.

Timeliness of Waits-for Graphs

In either method described above, the union of the local waits-for graphs that any particular site knows about currently does not have to reflect the situation that existed globally at any particular time. That doesn’t prevent the detection of deadlocks, since if a cycle in the global waits-for graph exists, it won’t go away until the deadlock is resolved by aborting at least one of the transactions involved in the cycle. Thus, the arcs of a cycle in the global graph will eventually all reach the central node (in method 1) or reach some node (in method 2), and the deadlock will be detected.

However, errors in the opposite direction can occur. There can be *phantom* deadlocks which appear as cycles in the union of the local waits-for graphs that have accumulated at some site, yet at no time did the global waits-for graph have this cycle.

Example 10.8: The transaction T_2 in Example 10.7 might decide to abort for one of several reasons, shortly after the local graph of Figure 10.7(a) was sent to the central site. Then the graph of Figure 10.7(b) might be sent to the central site. Before an update to Figure 10.7(a) can reach the central site, that node constructs the graph of Figure 10.7(c). Thus, it appears that there is a deadlock, and the central node will select a victim to abort. If it selects T_2 , there is no harm, since T_2 aborted anyway. However, it could just as well select T_1 , which would waste resources. \square

Timestamp-Based Deadlock Prevention

We mentioned schemes that avoid deadlocks by controlling the order in which

¹⁶ Note that in comparison, centralized or coordinator-based distributed commit protocols

items are locked by any given transaction, e.g., locking in lexicographic order or taking all locks at once. There also are schemes that do not place constraints on the order in which items are locked or accessed, but still can assure no deadlocks occur. These schemes use timestamps on transactions, and each guarantees that no cycles can occur in the global waits-for graph. It is important to note that the timestamps are used for deadlock avoidance only; access control of items is still by locking.

In one scheme, should (a subtransaction of) T_1 be waiting for (a subtransaction of) T_2 , then it must be that the timestamp of T_1 is less than the timestamp of T_2 ; in the second scheme, the opposite is true. In either scheme, a cycle in the waits-for graph would consist of transactions with monotonically increasing or monotonically decreasing timestamps, as we went around the cycle. Neither is possible, since when we go around the cycle we come back to the same timestamp that we started with.

We now define the two deadlock avoidance schemes. Suppose we have transactions T_1 and T_2 with timestamps t_1 and t_2 , respectively, and a subtransaction of T_1 attempts to access an item A locked by a subtransaction of T_2 .

- 1. In the *wait-die* scheme, T_1 waits for a lock on A if $t_1 < t_2$, i.e., if T_1 is the older transaction. If $t_1 > t_2$, then T_1 is aborted.
- 2. In the *wound-wait* scheme, T_1 waits for a lock on A if $t_1 > t_2$. If $t_1 < t_2$, then T_2 is forced to abort and release its lock on A to T_1 .¹⁷

In either scheme, the aborted transaction must initiate again with the same timestamp, not with a new timestamp. Reusing the original timestamp guarantees that the oldest transaction, in either scheme, cannot die or be wounded. Thus, each transaction will eventually be allowed to complete, as the following theorem shows.

Theorem 10.3: There can be neither deadlocks nor livelocks in the wait-die or the wound-wait schemes.

Proof: Consider the wait-die scheme. Suppose there is a cycle in the global waits-for graph, i.e., a sequence of transactions T_1, \dots, T_k such that each T_i is waiting for release of a lock by T_{i+1} , for $1 \leq i < k$, and T_k is waiting for T_1 . Let t_i be the timestamp of T_i . Then $t_1 < t_2 < \dots < t_k < t_1$, which implies $t_1 < t_1$, an impossibility. Similarly, in the wound-wait scheme, such a cycle would imply $t_1 > t_2 > \dots > t_k > t_1$, which is also impossible.

To see why no livelocks occur, let us again consider the wait-die scheme. If

¹⁷ Incidentally, the term “wound-wait” rather than “kill-wait” is used because of the image that the “wounded” subtransaction must, before it dies, run around informing all the other subtransactions of its transaction that they too must abort. That is not really necessary if a distributed commit algorithm is used, but the subject is gruesome, and the less said the better.

Method	Messages	Phantom aborts	Other
Timeout	None	Medium number	Can abort more than one trans- action to resolve one deadlock
Waits-for Graph Centralized	Medium traffic	Few	Vulnerable to node failure, bottlenecks
Waits-for Graph Distributed	High traffic	Few	
Timestamp	None	Many	

Figure 10.8 Comparison of deadlock-handling methods.

T is the transaction with the lowest timestamp, that is, T is the oldest transaction that has not completed, then T never dies. It may wait for younger transactions to release their locks, but since there are no deadlocks, those locks will eventually be released, and T will eventually complete. When T first initiates, there are some finite number of live, older transactions. By the argument above, each will eventually complete, making T the oldest. At that point, T is sure to complete the next time it is restarted. Of course, in ordinary operation, transactions will not necessarily complete in the order of their age, and in fact most will proceed without having to abort.

The no-livelock argument for the wound-wait scheme is similar. Here, the oldest transaction does not even have to wait for others to release locks; it takes the locks it needs and wounds the transactions holding them. \square

Comparison of Methods

Figure 10.8 summarizes the advantages and disadvantages of the methods we have covered in this section. The column labeled “Messages” refers to the message traffic needed to detect deadlocks. The column “Phantom aborts” refers to the possibility that transactions not involved in a deadlock will be required to abort.