In case (a), M must remember that N has asked it for the token, but does not know whether it can have it yet [another site could answer (b)]. M "reserves" the token for N; doing so prevents another site P from also being told by M that it has no objection to P's obtaining the token.[4]

3. If all sites reply (a) to N, then N knows it can have the write-token. It sends a message to each site that replied (a), telling it that N has accepted the write-token, and they should destroy whatever tokens they have for A. If some site replies (b), then N cannot have the write-token, and it must send messages to the nodes that replied (a) telling them they can cease reserving the write-token for A, and may allow another site to get that token.

To read A, essentially the same process takes place, except that if the local site has any of the read-tokens for A, no messages need to be sent. In (2) above, the responding site M does not object [send message (b)] if it has a read-token for A, only if it has a write-token. In (3), if N is allowed to obtain a read-token for A, then only write-tokens, not read-tokens, are destroyed at other sites.

More Comparisons Among Methods

Evidently, the primary copy token method uses considerably more messages than the other methods so far; both reading and writing can use $3m$ control messages, where m is number of nodes in the network, while other methods use a number of messages that is proportional to the number of copies of an item, at worst. On the other hand, the primary copy token approach averages much less than $3m$ control messages per lock operation when one site runs most of the transactions that reference a particular item. Then the write-token for that item will tend to reside at that site, making control messages unneeded for most transactions. Thus, a direct comparison with the k-of-n methods is not possible; which is preferable depends on the site distribution of the transactions that lock a particular item.

Similarly, we cannot compare the primary site method directly with the write-locks-all method; while the former uses smaller numbers of messages on the average, the latter has the advantage when most locks are read-locks on copies that are not at the primary site for that item. It appears that the primary site approach is more efficient than the k-of-n methods for $k > 1$. However, there are other considerations that might enter into the picture. For example, the primary site method is vulnerable to a failure at the primary site

[4] The reason we must be careful is that there might be no tokens for A at all. For example, none might have been created, or the last one could have been lost, because the node holding it failed. If we did not use "reservations," two sites could ask for the write-token for A at the same time, and each be told by all of the sites (including each other) that they did not have any token on A. Then, each would create a write-token for A and there would be two tokens when at most one should exist.

for an item, as the sites must then detect the failure and send messages to agree on a new primary site. In comparison, k-of-n type strategies can continue locking that item with no interruption.

We can also compare primary copy token methods with the primary site approach. In the later method, a write requires two control messages to request and receive a lock from the primary site, then n data messages, as usual, to write the new value. Reading requires a control message asking for a lock and a data message in response, granting the request and sending the value. If all transactions referencing A run at the primary site for A, then the two approaches are exactly the same; no messages are sent, except for the obligatory writes to update other copies of A, if any. When other sites do reference A, the primary site method appears to save a considerable number of messages.

However, the token method is somewhat more adaptable to temporary changes in behavior. For example, in a hypothetical bank database, suppose a customer goes on vacation and starts using a branch different from his usual one. Under the primary site method, each transaction at the new branch would require an exchange of locking messages. In comparison, under the token approach, after the first transaction ran at the new branch, the write-token for the account would reside at that branch as long as the customer was on vacation.

The Central Node Method

The last approach to locking that we shall consider is that in which one particular node of the network is given the responsibility for all locking. This method is almost like the primary site method; the only difference is that the primary site for an item, being the one *central* node, may not be a site that has a copy of the item. Thus, a read-lock must be garnered by the following steps:

1. Request a read-lock from the central node.
2. If not granted, the central node sends a message to the requesting site to that effect. If granted, the central node sends a message to a site with a copy of the item.
3. The site with the copy sends a message with the value to the requesting site.

Hence, the central node method often requires an extra control message to tell some other site to ship the value desired. Similarly, when writing, the site running the transaction must often send an extra message to the central node telling it to release the lock. In the primary site method, this message would be included with the messages committing the transaction.

Therefore, it seems that the central node approach behaves almost like the primary site method, but slower. Moreover, while it does not show in our model, which only counts messages without regard for destination, there is the added disadvantage that most of the message traffic is headed to or from one node, thus creating a potential bottleneck. Additionally, this method is especially vulnerable to a crash of the central node.

However, the algorithm has its redeeming features, also in areas not covered by our model. For example, under certain assumptions about loads on the system, there is an advantage to be had by bundling messages to and from the central site. The case for the central node approach is made by Garcia-Molina [1979].

Summary

The relative merits and demerits of the various approaches are summarized in Figure 10.2. We use n for the number of copies of an item and m for the total number of nodes. We assume in each case that the lock is granted and we ignore the possible savings that result if we can read or write at the same site as the transaction, thus saving a data message. The tabulation of Figure 10.2 counts only control messages, since each write requires n data messages, and each read requires one data message, no matter what the locking method.

Method	Control Msgs. to Write	Control Msgs. to Read	Comments
Write-Locks-All	$2n$	1	Good if read dominates
Majority	$\geq n+1$	$\geq n$	Avoids some deadlock
Primary Site	2	1	Efficient; some vulnerability to crash
Primary Copy Token	0–$4m$	0–$4m$	Adapts to changes in use pattern
Central Node	3	2	Vulnerable to crash; efficiencies may result from centralized traffic pattern

Figure 10.2 Advantages and disadvantages of distributed locking methods.

10.3 DISTRIBUTED TWO-PHASE LOCKING

From the last section, we see that it is feasible to define locks on logical items in various ways. Now, we must consider how to use locking to ensure the serializability of transactions that consist of several subtransactions, each running at a different site. Recall that a schedule of transactions in a distributed environment is a sequence of events, each occurring at one site. While several sites may perform actions simultaneously, we shall break ties arbitrarily, and assume that, according to some global clock, there is a linear order to events. A schedule is *serial* if it consists of all the actions for one transaction, followed by all the actions for another, and so on. A schedule is *serializable* if it is equivalent, in its effect on the database, to a serial schedule.

Recalling the strong relationship between serializability and two-phase locking from Section 9.3, let us consider how two-phase locking can be generalized to the distributed environment. Our first guess might be that at each node, the subtransactions should follow the two-phase protocol. However, that is not enough, as the following example shows.

Example 10.3: Suppose that logical transaction T_1 has two subtransactions:

1. $T_{1.1}$, which runs at site S_1 and writes a new value for copy A_1 of logical item A, and
2. $T_{1.2}$, which runs at site S_2 and writes the same new value for copy A_2 of A.

Also, transaction T_2 has two subtransactions, $T_{2.1}$ running at S_1 and writing a new value of A_1, and $T_{2.2}$, running at S_2 and writing the same value into A_2. We shall assume that write-locks-all is the protocol followed by these transactions for defining locks on logical items, but as we shall see, other methods cause similar problems.

$T_{1.1}$	$T_{2.1}$	$T_{1.2}$	$T_{2.2}$
WLOCK A_1			WLOCK A_2
UNLOCK A_1			UNLOCK A_2
	WLOCK A_1	WLOCK A_2	
	UNLOCK A_1	UNLOCK A_2	
At S_1		At S_2	

Figure 10.3 Transactions with two-phase locking at each node.

For the example at hand, we see in Figure 10.3 a possible schedule of actions at the two sites. Pairs of events on each line could occur simultaneously, or we could assume they occur in either order; it doesn't matter. Evidently, the situation at site S_1 tells us that $T_{1.1}$ must precede $T_{2.1}$ in the serial order. At

S_2 we find that $T_{2.2}$ must precede $T_{1.2}$. Unfortunately, a serial order must be formed not just from the subtransactions, but from (logical) transactions. Thus, if we choose to have T_1 precede T_2, then $T_{1.2}$ precedes $T_{2.2}$, violating the local ordering at S_2. Similarly, if the serial order is T_2, T_1, then the local ordering at S_1 is violated. In fact, in the order of events indicated in Figure 10.3, the two copies of A receive different final values, which should immediately convince us that no equivalent serial order exists.

The problem indicated above is not restricted to write-locks-all. For example, suppose we use the primary site method of locking. We can modify Figure 10.3 by letting A_1 be the sole copy of A and letting A_2 be the sole copy of another logical item B. Therefore, S_1 and S_2 are the primary sites for A and B, respectively. The schedule of Figure 10.3 is still not serializable, since the final value of B is that written by T_1 and the final value of A is what T_2 writes. In fact, notice that all the locking methods of Section 10.2 become the same when there is only one copy of each item; thus this problem of nonserializability comes up no matter what method we use. $\square$

Strict Two-Phase Locking

The problem illustrated by Example 10.3 is that in order for distributed transactions to behave as if they are two-phase locked, we must consider not only the local schedules, but the global schedule of actions, and that schedule must be two-phase locked. The consequence is that a subtransaction of T cannot release any lock if it is possible that another subtransaction of T at another site will later request a lock. For example, $T_{1.1}$ of Figure 10.3 violated this principle by unlocking A_1 before $T_{1.2}$ got its lock on A_2.

Thus, each subtransaction of a given transaction must inform the other subtransactions that it has requested all of its locks. Only after all subtransactions have reached their individual lock points has the transaction as a whole reached its lock point, after which the subtransactions may release their locks. The problem of all subtransactions agreeing that they have reached the lock point is one example of a *distributed agreement problem.* We shall study another, the distributed agreement to commit, in the next section. It will then become clear that distributed agreement, especially in the face of possible network failures, is very complex and expensive. Thus, the sending of control messages to establish that the subtransactions have reached their lock points is not normally sensible.

Rather, there are many reasons to insist that transactions in a distributed environment be strict, that is, they unlock only after reaching their commit point. For example, Section 9.8 discussed the problem of reading dirty data and consequent cascading rollback, e.g., which strict two-phase locking solves. If our transactions obey the strict protocol, then we can use the commit point as the lock point. The subtransactions agree to commit, by a process described

in the next section, and only after committing are locks released.

In a situation like Figure 10.3, $T_{1.1}$ and $T_{2.2}$ would not release their locks at the second line, if the strict protocol were followed. In this case, there would be a deadlock between T_1 and T_2, since each has a subtransaction that is waiting for a lock held by a subtransaction of the other. We shall discuss distributed deadlock detection in Section 10.8. In this case, one of T_1 and T_2 has to abort, along with all of its subtransactions.

10.4 DISTRIBUTED COMMITMENT

For the reason just discussed (supporting distributed two-phase locking), as well as for the reasons discussed in Sections 9.8 and 9.10 (resiliency), it is necessary for a distributed transaction to perform a commit action just before termination. The existence of subtransactions at various sites complicates the process considerably.

Suppose we have a transaction T which initiated at one site and spawned subtransactions at several other sites. We shall call the part of T that executes at its home site a subtransaction of the logical transaction T; thus logical T consists solely of subtransactions, each executing at a different site. We distinguish the subtransaction at the home site by calling it the *coordinator*, while the other subtransactions are the *participants.* This distinction is important when we describe the distributed commitment process.

In the absence of failures, distributed commitment is conceptually simple. Each subtransaction T_i of logical transaction T decides whether to commit or abort. Recall, T_i could abort for any of the reasons discussed in Chapter 9, such as involvement in a deadlock or an illegal database access. When T_i decides what it wants to do, it sends a `vote-commit` or `vote-abort` message to the coordinator. If the `vote-abort` message is sent, T_i knows the logical transaction T must abort, and therefore T_i may terminate. However, if T_i sends the `vote-commit` message, it does not know whether T will eventually commit, or whether some other subtransaction will decide to abort, thus causing T to abort.

Thus, after voting to commit, T_i must wait for a message from the coordinator. If the coordinator receives a `vote-abort` message from any subtransaction, it sends `abort` messages to all of the subtransactions, and they all abort, thus aborting the logical transaction T. If the coordinator receives `vote-commit` messages from all subtransactions (including itself), then it knows that T may commit. The coordinator sends `commit` messages to all of the subtransactions. Now, the subtransactions all know that T can commit, and they take what steps are necessary at their local site to perform the commitment, e.g., writing in the log and releasing locks.

It is useful to visualize the subtransactions changing state in response to their changes in knowledge about the logical transaction. In Figure 10.4, the