!! **Exercise 20.4.6:** Not all binary operations on relations located at different nodes of a network can have their execution time reduced by preliminary operations like the semijoin. Is it possible to improve on the obvious algorithm (ship one of the relations to the other site) when the operation is (a) union (b) intersection (c) difference?

20.5 Distributed Commit

In this section, we shall address the problem of how a distributed transaction that has components at several sites can execute atomically. The next section discusses another important property of distributed transactions: executing them serializably.

20.5.1 Supporting Distributed Atomicity

We shall begin with an example that illustrates the problems that might arise.

Example 20.13: Consider our example of a chain of stores mentioned in Section 20.3. Suppose a manager of the chain wants to query all the stores, find the inventory of toothbrushes at each, and issue instructions to move toothbrushes from store to store in order to balance the inventory. The operation is done by a single global transaction T that has component T_i at the ith store and a component T_0 at the office where the manager is located. The sequence of activities performed by T are summarized below:

1. Component T_0 is created at the site of the manager.

2. T_0 sends messages to all the stores instructing them to create components T_i.

3. Each T_i executes a query at store i to discover the number of toothbrushes in inventory and reports this number to T_0.

4. T_0 takes these numbers and determines, by some algorithm we do not need to discuss, what shipments of toothbrushes are desired. T_0 then sends messages such as "store 10 should ship 500 toothbrushes to store 7" to the appropriate stores (stores 7 and 10 in this instance).

5. Stores receiving instructions update their inventory and perform the shipments.

□

There are a number of things that could go wrong in Example 20.13, and many of these result in violations of the atomicity of T. That is, some of the actions comprising T get executed, but others do not. Mechanisms such as logging and recovery, which we assume are present at each site, will assure that each T_i is executed atomically, but do not assure that T itself is atomic.

Example 20.14: Suppose a bug in the algorithm to redistribute toothbrushes might cause store 10 to be instructed to ship more toothbrushes than it has. T_{10} will therefore abort, and no toothbrushes will be shipped from store 10; neither will the inventory at store 10 be changed. However, T_7 detects no problems and commits at store 7, updating its inventory to reflect the supposedly shipped toothbrushes. Now, not only has T failed to execute atomically (since T_{10} never completes), but it has left the distributed database in an inconsistent state. □

Another source of problems is the possibility that a site will fail or be disconnected from the network while the distributed transaction is running.

Example 20.15: Suppose T_{10} replies to T_0's first message by telling its inventory of toothbrushes. However, the machine at store 10 then crashes, and the instructions from T_0 are never received by T_{10}. Can distributed transaction T ever commit? What should T_{10} do when its site recovers? □

20.5.2 Two-Phase Commit

In order to avoid the problems suggested in Section 20.5.1, distributed DBMS's use a complex protocol for deciding whether or not to commit a distributed transaction. In this section, we shall describe the basic idea behind these protocols, called *two-phase commit.*[5] By making a global decision about committing, each component of the transaction will commit, or none will. As usual, we assume that the atomicity mechanisms at each site assure that either the local component commits or it has no effect on the database state at that site; i.e., components of the transaction are atomic. Thus, by enforcing the rule that either all components of a distributed transaction commit or none does, we make the distributed transaction itself atomic.

Several salient points about the two-phase commit protocol follow:

- In a two-phase commit, we assume that each site logs actions at that site, but there is no global log.
- We also assume that one site, called the *coordinator*, plays a special role in deciding whether or not the distributed transaction can commit. For example, the coordinator might be the site at which the transaction originates, such as the site of T_0 in the examples of Section 20.5.1.
- The two-phase commit protocol involves sending certain messages between the coordinator and the other sites. As each message is sent, it is logged at the sending site, to aid in recovery should it be necessary.

With these points in mind, we can describe the two phases in terms of the messages sent between sites.

[5]Do not confuse two-phase commit with two-phase locking. They are independent ideas, designed to solve different problems.

Phase I

In phase 1 of the two-phase commit, the coordinator for a distributed transaction T decides when to attempt to commit T. Presumably the attempt to commit occurs after the component of T at the coordinator site is ready to commit, but in principle the steps must be carried out even if the coordinator's component wants to abort (but with obvious simplifications as we shall see). The coordinator polls the sites of all components of the transaction T to determine their wishes regarding the commit/abort decision, as follows:

1. The coordinator places a log record <`Prepare` T> on the log at its site.

2. The coordinator sends to each component's site (in principle including itself) the message `prepare` T.

3. Each site receiving the message `prepare` T decides whether to commit or abort its component of T. The site can delay if the component has not yet completed its activity, but must eventually send a response.

4. If a site wants to commit its component, it must enter a state called *precommitted.* Once in the precommitted state, the site cannot abort its component of T without a directive to do so from the coordinator. The following steps are done to become precommitted:

   (a) Perform whatever steps are necessary to be sure the local component of T will not have to abort, even if there is a system failure followed by recovery at the site. Thus, not only must all actions associated with the local T be performed, but the appropriate actions regarding the log must be taken so that T will be redone rather than undone in a recovery. The actions depend on the logging method, but surely the log records associated with actions of the local T must be flushed to disk.

   (b) Place the record <`Ready` T> on the local log and flush the log to disk.

   (c) Send to the coordinator the message `ready` T.

   However, the site does not commit its component of T at this time; it must wait for phase 2.

5. If, instead, the site wants to abort its component of T, then it logs the record <`Don't commit` T> and sends the message `don't commit` T to the coordinator. It is safe to abort the component at this time, since T will surely abort if even one component wants to abort.

The messages of phase 1 are summarized in Fig. 20.12.

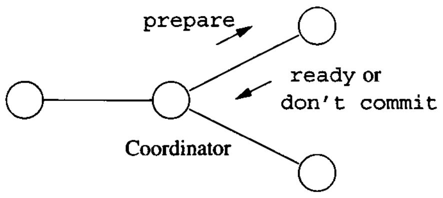

Figure 20.12: Messages in phase 1 of two-phase commit

Phase II

The second phase begins when responses `ready` or `don't commit` are received from each site by the coordinator. However, it is possible that some site fails to respond; it may be down, or it has been disconnected by the network. In that case, after a suitable timeout period, the coordinator will treat the site as if it had sent `don't commit`.

1. If the coordinator has received `ready` T from all components of T, then it decides to commit T. The coordinator logs <`Commit` T> at its site and then sends message `commit` T to all sites involved in T.

2. However, if the coordinator has received `don't commit` T from one or more sites, it logs <`Abort` T> at its site and then sends `abort` T messages to all sites involved in T.

3. If a site receives a `commit` T message, it commits the component of T at that site, logging <`Commit` T> as it does.

4. If a site receives the message `abort` T, it aborts T and writes the log record <`Abort` T>.

The messages of phase 2 are summarized in Fig. 20.13.

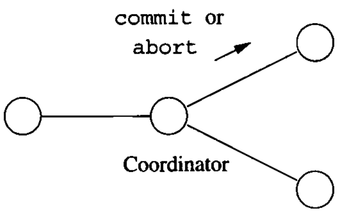

Figure 20.13: Messages in phase 2 of two-phase commit

20.5.3 Recovery of Distributed Transactions

At any time during the two-phase commit process, a site may fail. We need to make sure that what happens when the site recovers is consistent with the global decision that was made about a distributed transaction T. There are several cases to consider, depending on the last log entry for T.

1. If the last log record for T was `<Commit` T`>`, then T must have been committed by the coordinator. Depending on the log method used, it may be necessary to redo the component of T at the recovering site.

2. If the last log record is `<Abort` T`>`, then similarly we know that the global decision was to abort T. If the log method requires it, we undo the component of T at the recovering site.

3. If the last log record is `<Don't commit` T`>`, then the site knows that the global decision must have been to abort T. If necessary, effects of T on the local database are undone.

4. The hard case is when the last log record for T is `<Ready` T`>`. Now, the recovering site does not know whether the global decision was to commit or abort T. This site must communicate with at least one other site to find out the global decision for T. If the coordinator is up, the site can ask the coordinator. If the coordinator is not up at this time, some other site may be asked to consult its log to find out what happened to T. In the worst case, no other site can be contacted, and the local component of T must be kept active until the commit/abort decision is determined.

5. It may also be the case that the local log has no records about T that come from the actions of the two-phase commit protocol. If so, then the recovering site may unilaterally decide to abort its component of T, which is consistent with all logging methods. It is possible that the coordinator already detected a timeout from the failed site and decided to abort T. If the failure was brief, T may still be active at other sites, but it will never be inconsistent if the recovering site decides to abort its component of T and responds with `don't commit` T if later polled in phase 1.

The above analysis assumes that the failed site is not the coordinator. When the coordinator fails during a two-phase commit, new problems arise. First, the surviving participant sites must either wait for the coordinator to recover or elect a new coordinator. Since the coordinator could be down for an indefinite period, there is good motivation to elect a new leader, at least after a brief waiting period to see if the coordinator comes back up.

The matter of *leader election* is in its own right a complex problem of distributed systems, beyond the scope of this book. However, a simple method will work in most situations. For instance, we may assume that all participant sites have unique identifying numbers, e.g., IP addresses. Each participant sends messages announcing its availability as leader to all the other sites, giving its identifying number. After a suitable length of time, each participant acknowledges as the new coordinator the lowest-numbered site from which it has heard, and sends messages to that effect to all the other sites. If all sites receive consistent messages, then there is a unique choice for new coordinator, and everyone knows about it. If there is inconsistency, or a surviving site has failed to respond, that too will be universally known, and the election starts over.

Now, the new leader polls the sites for information about each distributed transaction T. Each site reports the last record on its log concerning T, if there is one. The possible cases are:

1. Some site has `<Commit` T`>` on its log. Then the original coordinator must have wanted to send `commit` T messages everywhere, and it is safe to commit T.

2. Similarly, if some site has `<Abort` T`>` on its log, then the original coordinator must have decided to abort T, and it is safe for the new coordinator to order that action.

3. Suppose now that no site has `<Commit` T`>` or `<Abort` T`>` on its log, but at least one site does *not* have `<Ready` T`>` on its log. Then since actions are logged before the corresponding messages are sent, we know that the old coordinator never received `ready` T from this site and therefore could not have decided to commit. It is safe for the new coordinator to decide to abort T.

4. The most problematic situation is when there is no `<Commit` T`>` or `<Abort` T`>` to be found, but every surviving site has `<Ready` T`>`. Now, we cannot be sure whether the old coordinator found some reason to abort T or not; it could have decided to do so because of actions at its own site, or because of a `don't commit` T message from another failed site, for example. Or the old coordinator may have decided to commit T and already committed its local component of T. Thus, the new coordinator is not able to decide whether to commit or abort T and must wait until the original coordinator recovers. In real systems, the database administrator has the ability to intervene and manually force the waiting transaction components to finish. The result is a possible loss of atomicity, but the person executing the blocked transaction will be notified to take some appropriate compensating action.

20.5.4 Exercises for Section 20.5

! **Exercise 20.5.1:** Consider a transaction T initiated at a home computer that asks bank B to transfer \$10,000 from an account at B to an account at another bank C.

a) What are the components of distributed transaction T? What should the components at B and C do?

b) What can go wrong if there is not \$10,000 in the account at B?

c) What can go wrong if one or both banks' computers crash, or if the network is disconnected?

d) If one of the problems suggested in (c) occurs, how could the transaction resume correctly when the computers and network resume operation?

Exercise 20.5.2: In this exercise, we need a notation for describing sequences of messages that can take place during a two-phase commit. Let (i, j, M) mean that site i sends the message M to site j, where the value of M and its meaning can be P (prepare), R (ready), D (don't commit), C (commit), or A (abort). We shall discuss a simple situation in which site 0 is the coordinator, but not otherwise part of the transaction, and sites 1 and 2 are the components. For instance, the following is one possible sequence of messages that could take place during a successful commit of the transaction:

$$(0,1,P),\ (0,2,P),\ (2,0,R),\ (1,0,R),\ (0,2,C),\ (0,1,C)$$

a) Give an example of a sequence of messages that could occur if site 1 wants to commit and site 2 wants to abort.

! b) How many possible sequences of messages such as the above are there, if the transaction successfully commits?

! c) If site 1 wants to commit, but site 2 does not, how many sequences of messages are there, assuming no failures occur?

! d) If site 1 wants to commit, but site 2 is down and does not respond to messages, how many sequences are there?

!! **Exercise 20.5.3:** Using the notation of Exercise 20.5.2, suppose the sites are a coordinator and n other sites that are the transaction components. As a function of n, how many sequences of messages are there if the transaction successfully commits?

20.6 Distributed Locking

In this section we shall see how to extend a locking scheduler to an environment where transactions are distributed and consist of components at several sites. We assume that lock tables are managed by individual sites, and that the component of a transaction at a site can request locks on the data elements only at that site.

When data is replicated, we must arrange that the copies of a single element X are changed in the same way by each transaction. This requirement introduces a distinction between locking the *logical* database element X and locking one or more of the copies of X. In this section, we shall offer a cost model for distributed locking algorithms that applies to both replicated and nonreplicated data. However, before introducing the model, let us consider an obvious (and sometimes adequate) solution to the problem of maintaining locks in a distributed database — centralized locking.