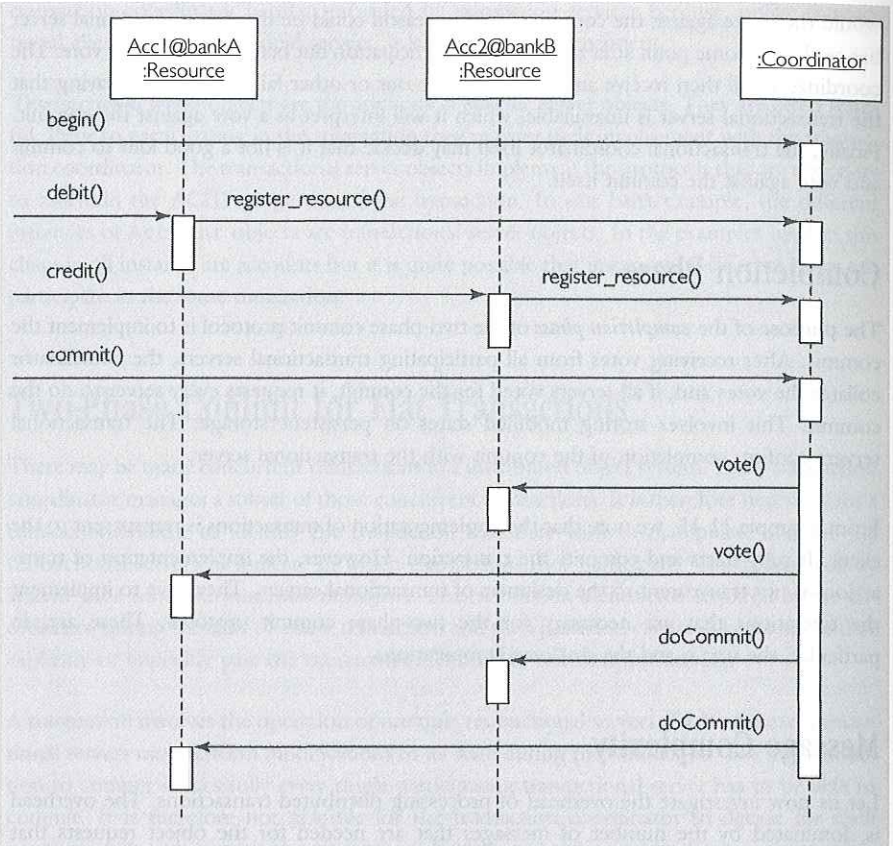
Example 11.15

Object Requests for Two-Phase Commit



This sequence diagram shows the two-phase commit for a distributed funds transfer transaction between two accounts residing on the hosts of different banks. First the transactional client requests the begin of a new transaction from the transaction coordinator. Then the transactional client invokes the `debit` and `credit` operations from the two account objects that are transactional servers. The implementation of these two operations know that they are part of a transaction and they therefore register their transaction involvement with the coordinator. The client then requests execution of the commit. To implement the commit, the coordinator first asks the two account objects for their vote to commit. In this example, both transactional servers vote in favour of the commit and the coordinator requests the `doCommit` operation from them. The coordinator then returns control to the client, indicating that the transaction was completed successfully.

The server uncertainty has to be bridged by the transactional server. After having voted in favour of a commit, the server must be able to complete the transaction. This means, in fact, that the server has to cope with a situation when it crashes after returning the vote. This is usually done by storing the changes of the transaction on some temporary persistent storage, for example a log file, that is used to recover the transaction result during a restart of the server.

The transactional coordinator also has to store some information on persistent temporary storage after the decision has been reached to do the commit. In particular, it has to store the transaction identifier, the references of participating transactional servers and the decision whether to commit the transaction. The server will use this information after a recovery from a failure that happened after the collation of the votes to re-transmit the `doCommit` requests to participating transactional servers.

Recovery

Let us now discuss what can go wrong during a transaction in order to reveal how the two-phase commit protocol recovers from failures. The aim of this discussion is to show how 2PC achieves atomicity.

Failure prior to commit: If any object involved with a transaction (including the transactional client) fails prior to requesting the commit message, the transaction coordinator will eventually abort the transaction by requesting an explicit abort from participating servers.

Failure of server before voting: If any server fails prior to voting, the coordinator will interpret the absence of a vote from the server as a vote against a commit and abort the transaction.

Failure of coordinator during voting: If the coordinator fails before or during the vote, the transactions will never receive a `doCommit` request and eventually abort.

Failure of server after voting: If a server fails after it has voted in favour of committing, after restarting it will ask the coordinator about the commit decision for the transaction. If the decision was to commit, it will commit using the data it recovered from the temporary persistent storage; otherwise it will abort.

Failure of coordinator after first `doCommit`: If the coordinator fails after it has started the completion phase, the coordinator has to use the data it stored on temporary persistent storage to re-transmit the `doCommit` requests to participating transactional servers.

Summary

All the changes that were performed by transactional servers are performed either completely or not performed at all. In order to achieve this atomicity, both transactional servers and the transaction coordinator utilize temporary persistent storage. The request of the first `doCommit` operation from a transactional server marks a decision point. Before that point, the coordinator might undo all changes by sending abort messages. From that point forward, recovery is used to complete the transaction.


Two-phase commit achieves atomicity of transactions.

11.3.3 Open Distributed Transaction Processing

In the previous chapter, we suggested that database management systems (DBMS) may be used to implement persistence. DBMSs also support the concept of transactions, though these transactions were traditionally confined to data that is stored in databases that are under full control of the DBMS. If some transactional servers use a relational database, another server uses an object database and a third server uses files to achieve persistence, how