14

DISTRIBUTED TRANSACTIONS

This chapter introduces distributed transactions – those that involve more than one server. Distributed transactions may be either flat or nested.

An atomic commit protocol is a cooperative procedure used by a set of servers involved in a distributed transaction. It enables the servers to reach a joint decision as to whether a transaction can be committed or aborted. This chapter describes the two-phase commit protocol, which is the most commonly used atomic commit protocol.

The section on concurrency control in distributed transactions discusses how locking, timestamp ordering and optimistic concurrency control may be extended for use with distributed transactions.

The use of locking schemes can lead to distributed deadlocks. Distributed deadlock detection algorithms are discussed.

Servers that provide transactions include a recovery manager whose concern is to ensure that the effects of transactions on the objects managed by a server can be recovered when it is replaced after a failure. The recovery manager saves the objects in permanent storage together with intentions lists and information about the status of each transaction.

14.1 Introduction

In Chapter 13, we discussed flat and nested transactions that accessed objects at a single server. In the general case, a transaction, whether flat or nested, will access objects located in several different computers. We use the term *distributed transaction* to refer to a flat or nested transaction that accesses objects managed by multiple servers.

When a distributed transaction comes to an end, the atomicity property of transactions requires that either all of the servers involved commit the transaction or all of them abort the transaction. To achieve this, one of the servers takes on a coordinator role, which involves ensuring the same outcome at all of the servers. The manner in which the coordinator achieves this depends on the protocol chosen. A protocol known as the 'two-phase commit protocol' is the most commonly used. This protocol allows the servers to communicate with one another to reach a joint decision as to whether to commit or abort.

Concurrency control in distributed transactions is based on the methods discussed in Chapter 13. Each server applies local concurrency control to its own objects, which ensures that transactions are serialized locally. Distributed transactions must be serialized globally. How this is achieved varies as to whether locking, timestamp ordering or optimistic concurrency control is in use. In some cases, the transactions may be serialized at the individual servers, but at the same time a cycle of dependencies between the different servers may occur and a distributed deadlock arise.

Transaction recovery is concerned with ensuring that all the objects involved in transactions are recoverable. In addition to that, it guarantees that the values of the objects reflect all the changes made by committed transactions and none of those made by aborted ones.

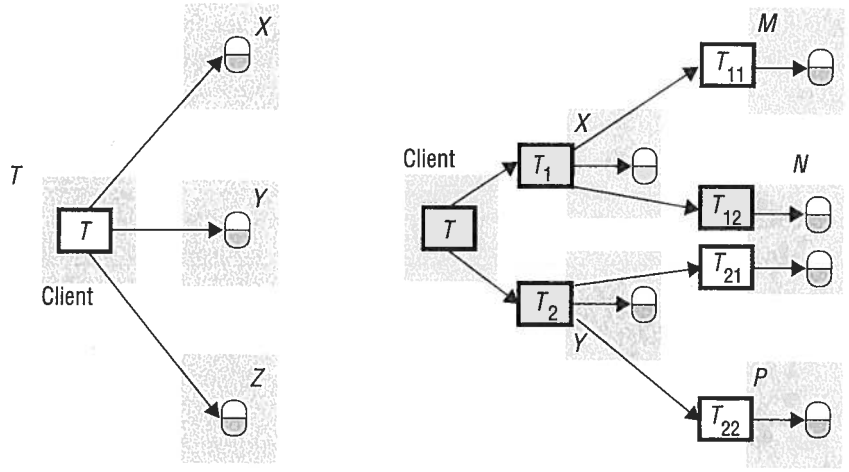
14.2 Flat and nested distributed transactions

A client transaction becomes distributed if it invokes operations in several different servers. There are two different ways that distributed transactions can be structured: as flat transactions and as nested transactions.

In a flat transaction, a client makes requests to more than one server. For example, in Figure 14.1(a), transaction T is a flat transaction that invokes operations on objects in servers X, Y and Z. A flat client transaction completes each of its requests before going on to the next one. Therefore, each transaction accesses servers' objects sequentially. When servers use locking, a transaction can only be waiting for one object at a time.

In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting. Figure 14.1(b) shows a client's transaction T that opens two subtransactions T_1 and T_2, which access objects at servers X and Y. The subtransactions T_1 and T_2 open further subtransactions T_{11}, T_{12}. T_{21} and T_{22}, which access objects at servers M, N and P. In the nested case, subtransactions at the same level can run concurrently, so T_1 and T_2 are concurrent, and as they invoke objects in different servers, they can run in parallel. The four subtransactions T_{11}, T_{12}, T_{21} and T_{22} also run concurrently.

Figure 14.1 Distributed transactions



Consider a distributed transaction in which a client transfers \$10 from account *A* to *C* and then transfers \$20 from *B* to *D*. Accounts *A* and *B* are at separate servers *X* and *Y* and accounts *C* and *D* are at server *Z*. If this transaction is structured as a set of four nested transactions, as shown in Figure 14.2, the four requests (two *deposit* and two *withdraw*) can run in parallel and the overall effect can be achieved with better performance than a simple transaction in which the four operations are invoked sequentially.

Figure 14.2 Nested banking transaction

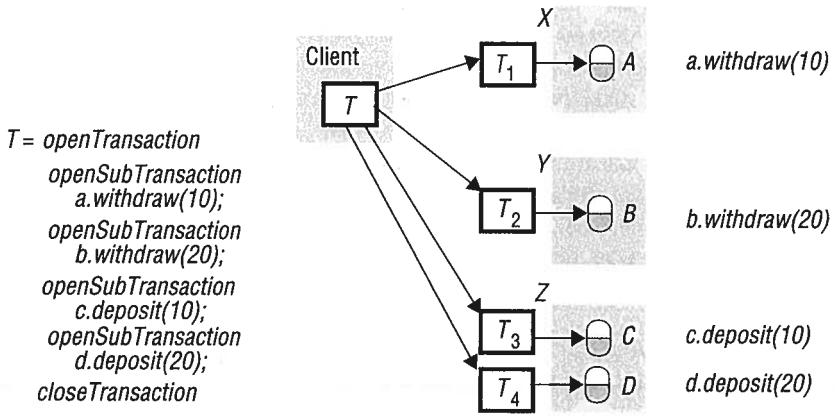
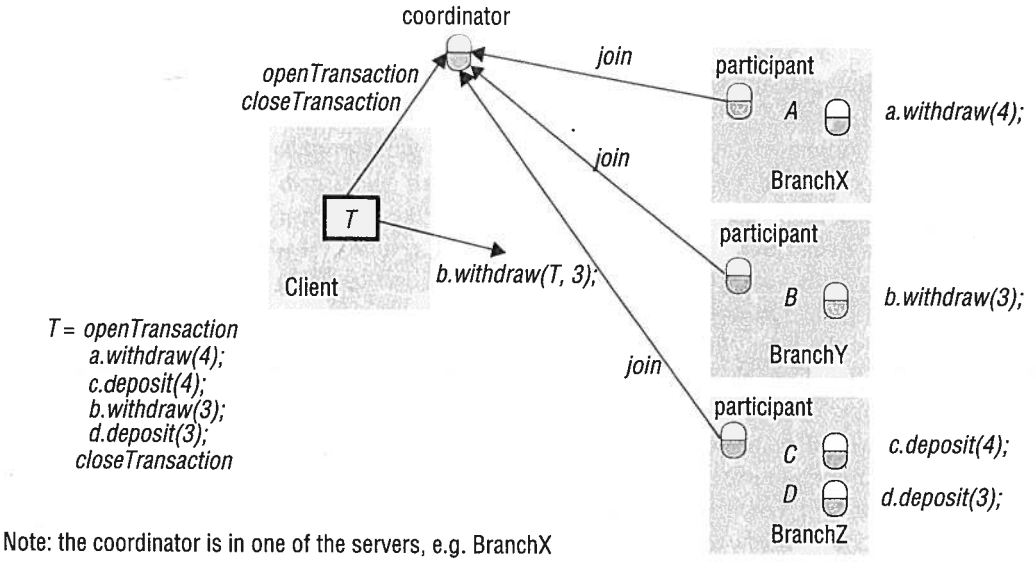


Figure 14.3 A distributed banking transaction



Note: the coordinator is in one of the servers, e.g. BranchX

14.2.1 The coordinator of a distributed transaction

Servers that execute requests as part of a distributed transaction need to be able to communicate with one another to coordinate their actions when the transaction commits. A client starts a transaction by sending an *openTransaction* request to a coordinator in any server, as described in Section 13.2. The coordinator that is contacted carries out the *openTransaction* and returns the resulting transaction identifier to the client. Transaction identifiers for distributed transactions must be unique within a distributed system. A simple way to achieve this is for a TID to contain two parts: the server identifier (for example, an IP address) of the server that created it and a number unique to the server.

The coordinator that opened the transaction becomes the *coordinator* for the distributed transaction and at the end is responsible for committing or aborting it. Each of the servers that manages an object accessed by a transaction is a participant in the transaction and provides an object we call the *participant*. Each participant is responsible for keeping track of all of the recoverable objects at that server involved in the transaction. The participants are responsible for cooperating with the coordinator in carrying out the commit protocol.

During the progress of the transaction, the coordinator records a list of references to the participants, and each participant records a reference to the coordinator.

The interface for *Coordinator* shown in Figure 13.3 provides an additional method, *join*, which is used whenever a new participant joins the transaction:

---

join(Trans, reference to participant)
Informs a coordinator that a new participant has joined the transaction *Trans*.

The coordinator records the new participant in its participant list. The fact that the coordinator knows all the participants and each participant knows the coordinator will enable them to collect the information that will be needed at commit time.

Figure 14.3 shows a client whose (flat) banking transaction involves accounts *A*, *B*, *C* and *D* at servers BranchX, BranchY and BranchZ. The client's transaction, *T*, transfers \$4 from account *A* to account *C* and then transfers \$3 from account *B* to account *D*. The transaction described on the left is expanded to show that *openTransaction* and *closeTransaction* are directed to the coordinator, which would be situated in one of the servers involved in the transaction. Each server is shown with a *participant*, which joins the transaction by invoking the *join* method in the coordinator. When the client invokes one of the methods in the transaction, for example *b.withdraw(T, 3)*, the object receiving the invocation (*B* at BranchY in this case) informs its participant object that the object belongs to the transaction *T*. If it has not already informed the coordinator, the participant object uses the *join* operation to do so. In this example, we show the transaction identifier being passed as an additional argument so that the recipient can pass it on to the coordinator. By the time the client calls *closeTransaction*, the coordinator has references to all of the participants.

Note that it is possible for a participant to call *abortTransaction* in the coordinator if for some reason it is unable to continue with the transaction.

14.3 Atomic commit protocols

Transaction commit protocols were devised in the early 1970s, and the two-phase commit protocol appeared in Gray [1978]. The atomicity of transactions requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them. In the case of a distributed transaction, the client has requested the operations at more than one server. A transaction comes to an end when the client requests that a transaction be committed or aborted. A simple way to complete the transaction in an atomic manner is for the coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they had carried it out. This is an example of a *one-phase atomic commit protocol.*

This simple one-phase atomic commit protocol is inadequate because, in the case when the client requests a commit, it does not allow a server to make a unilateral decision to abort a transaction. Reasons that prevent a server from being able to commit its part of a transaction generally relate to issues of concurrency control. For example, if locking is in use, the resolution of a deadlock can lead to the aborting of a transaction without the client being aware unless it makes another request to the server. If optimistic concurrency control is in use, the failure of validation at a server would cause it to decide to abort the transaction. The coordinator may not know when a server has crashed and been replaced during the progress of a distributed transaction – such a server will need to abort the transaction.

The *two-phase commit protocol* is designed to allow any participant to abort its part of a transaction. Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must also be aborted. In the first phase of the

protocol, each participant votes for the transaction to be committed or aborted. Once a participant has voted to commit a transaction, it is not allowed to abort it. Therefore, before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim. A participant in a transaction is said to be in a *prepared* state for a transaction if it will eventually be able to commit it. To make sure of this, each participant saves in permanent storage all of the objects that it has altered in the transaction, together with its status – prepared.

In the second phase of the protocol, every participant in the transaction carries out the joint decision. If any one participant votes to abort, then the decision must be to abort the transaction. If all the participants vote to commit, then the decision is to commit the transaction.

The problem is to ensure that all of the participants vote and that they all reach the same decision. This is fairly simple if no errors occur, but the protocol must work correctly even when some of the servers fail, messages are lost or servers are temporarily unable to communicate with one another.

Failure model for the commit protocols ◊ Section 13.1.2 presents a failure model for transactions that applies equally to the two-phase (or any other) commit protocol. Commit protocols are designed to work in an asynchronous system in which servers may crash and messages may be lost. It is assumed that an underlying request-reply protocol removes corrupt and duplicated messages. There are no byzantine faults – servers either crash or else they obey the messages they are sent.

The two-phase commit protocol is an example of a protocol for reaching a consensus. Chapter 12 asserts that consensus cannot be reached in an asynchronous system if processes sometimes fail. However, the two-phase commit protocol does reach consensus under those conditions. This is because crash failures of processes are masked by replacing a crashed process with a new process whose state is set from information saved in permanent storage and information held by other processes.

14.3.1 The two-phase commit protocol

During the progress of a transaction, there is no communication between the coordinator and the participants apart from the participants informing the coordinator when they join the transaction. A client's request to commit (or abort) a transaction is directed to the coordinator. If the client requests *abortTransaction*, or if the transaction is aborted by one of the participants, the coordinator informs the participants immediately. It is when the client asks the coordinator to commit the transaction that two-phase commit protocol comes into use.

In the first phase of the two-phase commit protocol the coordinator asks all the participants if they are prepared to commit; and in the second, it tells them to commit (or abort) the transaction. If a participant can commit its part of a transaction, it will agree as soon as it has recorded the changes and its status in permanent storage – and is prepared to commit. The coordinator in a distributed transaction communicates with the participants to carry out the two-phase commit protocol by means of the operations summarized in Figure 14.4. The methods *canCommit*, *doCommit* and *doAbort* are

Figure 14.4 Operations for two-phase commit protocol

canCommit?(trans)→ Yes / No
Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

doCommit(trans)
Call from coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)
Call from coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant)
Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) → Yes / No
Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

methods in the interface of the participant. The methods *haveCommitted* and *getDecision* are in the coordinator interface.

The two-phase commit protocol consists of a voting phase and a completion phase as shown in Figure 14.5. By the end of step (2) the coordinator and all the participants that voted *Yes* are prepared to commit. By the end of step (3) the transaction is effectively completed. At step (3a) the coordinator and the participants are committed, so the coordinator can report a decision to commit to the client. At (3b) the coordinator reports a decision to abort to the client.

At step (4) participants confirm that they have committed so that the coordinator knows when the information it has recorded about the transaction is no longer needed.

This apparently straightforward protocol could fail due to one or more of the servers crashing or due to a breakdown in communication between the servers. To deal with the possibility of crashing, each server saves information relating to the two-phase commit protocol in permanent storage. This information can be retrieved by a new process that is started to replace a crashed server. The recovery aspects of distributed transactions are discussed in Section 14.6.

The exchange of information between the coordinator and participants can fail when one of the servers crashes, or when messages are lost. Timeouts are used to avoid processes blocking for ever. When a timeout occurs at a process, it must take an appropriate action. To allow for this the protocol includes a timeout action for each step at which a process may block. These actions are designed to allow for the fact that in an asynchronous system, a timeout may not necessarily imply that a server has failed.

Timeout actions in the two-phase commit protocol ◊ There are various stages in the protocol at which the coordinator or a participant cannot progress its part of the protocol until it receives another request or reply from one of the others.

Figure 14.5 The two-phase commit protocol

Phase 1 (voting phase):

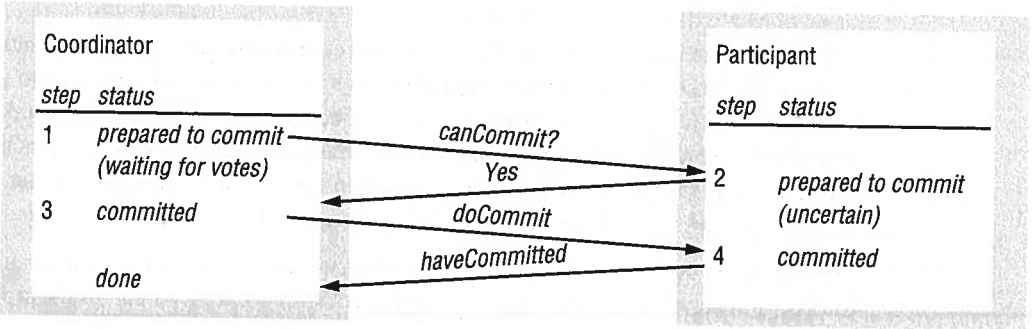
1. The coordinator sends a *canCommit?* request to each of the participants in the transaction.
2. When a participant receives a *canCommit?* request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Consider first the situation where a participant has voted *Yes* and is waiting for the coordinator to report on the outcome of the vote by telling it to commit or abort the transaction. See step (2) in Figure 14.6. Such a participant is *uncertain* of the outcome and cannot proceed any further until it gets the outcome of the vote from the coordinator. The participant cannot decide unilaterally what to do next, and meanwhile the objects used by its transaction cannot be released for use by other transactions. The participant makes a *getDecision* request to the coordinator to determine the outcome of the transaction. When it gets the reply it continues the protocol at step (4) in Figure 14.5. If

Figure 14.6 Communication in two-phase commit protocol



the coordinator has failed, the participant will not be able to get the decision until the coordinator is replaced, which can result in extensive delays for participants in the uncertain state.

Other alternative strategies are available for the participants to obtain a decision cooperatively instead of contacting the coordinator. These strategies have the advantage that they may be used when the coordinator has failed. See Exercise 14.5 and Bernstein *et al.* [1987] for details. However, even with a cooperative protocol, if all the participants are in the *uncertain* state, they will be unable to get a decision until the coordinator or a participant with the knowledge is available.

Another point at which a participant may be delayed is when it has carried out all its client requests in the transaction but has not yet received a *canCommit*? call from the coordinator. As the client sends the *closeTransaction* to the coordinator, a participant can only detect such a situation if it notices that it has not had a request in a particular transaction for a long time, for example by a timeout period on a lock. As no decision has been made at this stage, the participant can decide to *abort* unilaterally after some period of time.

The coordinator may be delayed when it is waiting for votes from the participants. As it has not yet decided the fate of the transaction it may decide to abort the transaction after some period of time. It must then announce *doAbort* to the participants who have already sent their votes. Some tardy participants may try to vote *Yes* after this, but their votes will be ignored and they will enter the *uncertain* state as described above.

Performance of the two-phase commit protocol ◊ Provided that all goes well – that is, that the coordinator and participants and the communication between them do not fail, the two-phase commit protocol involving N participants can be completed with N *canCommit?* messages and replies, followed by N *doCommit* messages. That is, the cost in messages is proportional to $3N$, and the cost in time is three rounds of messages. The *haveCommitted* messages are not counted in the estimated cost of the protocol, which can function correctly without them – their role is to enable servers to delete stale coordinator information.

In the worst case, there may be arbitrarily many server and communication failures during the two-phase commit protocol. However, the protocol is designed to tolerate a succession of failures (server crashes or lost messages) and is guaranteed to complete eventually, although it is not possible to specify a time limit within which it will be completed.

As noted in the section on timeouts, the two-phase commit protocol can cause considerable delays to participants in the *uncertain* state. These delays occur when the coordinator has failed and cannot reply to *getDecision* requests from participants. Even if a cooperative protocol allows participants to make *getDecision* requests to other participants, delays will occur if all the active participants are *uncertain*.

Three-phase commit protocols have been designed to alleviate such delays. They are more expensive in the number of messages and the number of rounds required for the normal (failure-free) case. For a description of three-phase commit protocols, see Exercise 14.2 and Bernstein *et al.* [1987].