MODULE 5

17.1 Introduction

In Chapter 16, 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 16. Each server applies local concurrency control to its own objects, which ensures that transactions are serialized locally. But distributed transactions must also be serialized globally. How this is achieved varies depending upon whether locking, timestamp ordering or optimistic concurrency control is in use. In some cases, the transactions may be serialized at the individual servers, but 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.

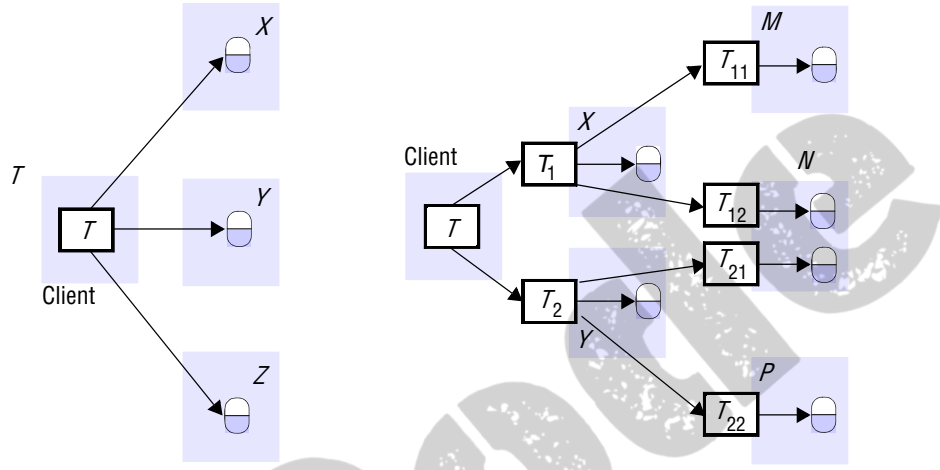
17.2 Flat and nested distributed transactions

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 17.1(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 17.1(b) shows a client 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 17.1 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 17.2, the four requests (two *deposits* and two *withdraws*) 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 17.2 Nested banking transaction

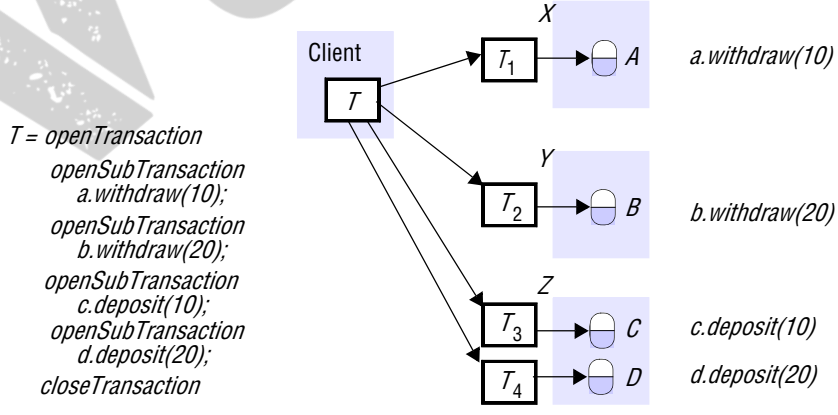
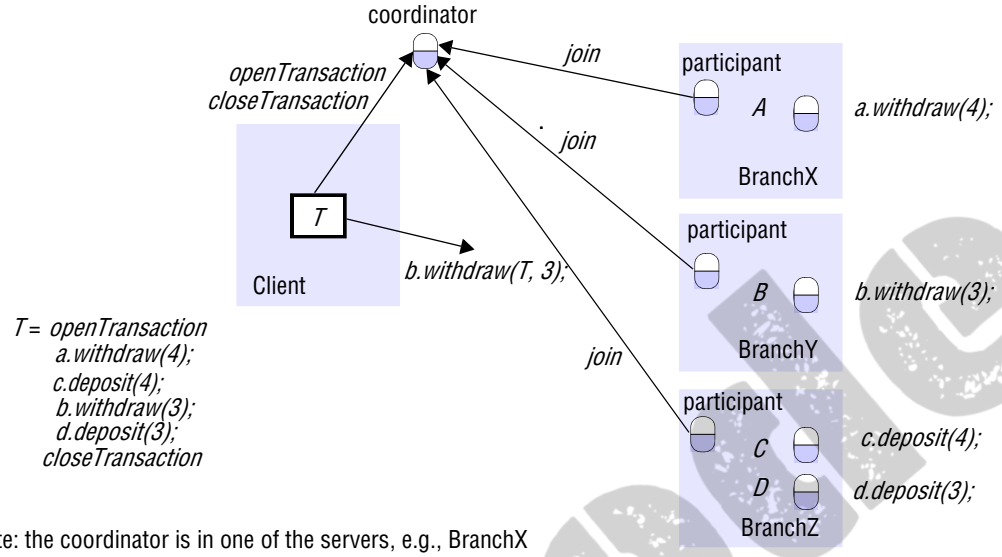


Figure 17.3 A distributed banking transaction



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

17.2.1 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 16.2. The coordinator that is contacted carries out the *openTransaction* and returns the resulting transaction identifier (TID) 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 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 that are 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 17.3 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.

17.3 Atomic commit protocols

Transaction commit protocols were devised in the early 1970s, and the two-phase commit protocol appeared in Gray [1978]. The atomicity property 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 operations at more than one server. A transaction comes to an end when the client requests that it 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 have carried it out. This is an example of a *one-phase atomic commit protocol*.

This simple one-phase atomic commit protocol is inadequate, though, because it does not allow a server to make a unilateral decision to abort a transaction when the client requests a commit. 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. Also if optimistic concurrency control is in use, the failure of validation at a server would cause it to decide to abort the transaction. Finally, the coordinator may not know if 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 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 16.1.2 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 obey the messages they are sent.

The two-phase commit protocol is an example of a protocol for reaching a consensus. Chapter 15 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.

17.3.1 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 all participants immediately. It is when the client asks the coordinator to commit the transaction that the 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; 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 it has made (to the objects) and its status in

Figure 17.4 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 when it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

permanent storage and is therefore 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 17.4. The methods *canCommit*, *doCommit* and *doAbort* are 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 17.5. 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 17.6.

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 forever. 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.

Figure 17.5 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.

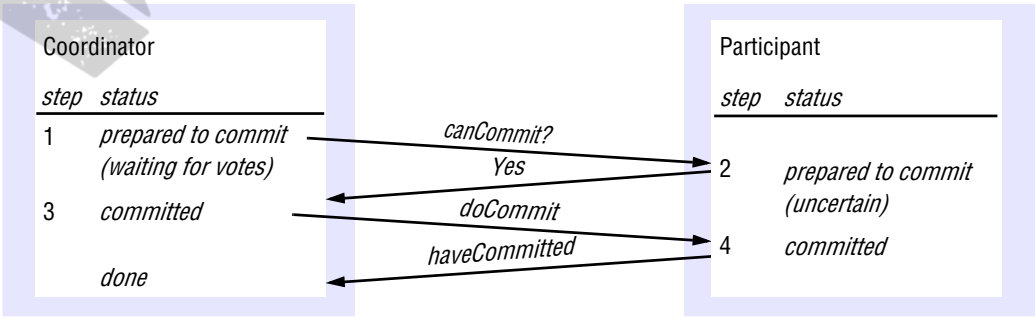
Phase 2 (completion according to outcome of vote):

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.

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.

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 (step 2 in Figure 17.6). 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

Figure 17.6 Communication in two-phase commit protocol



used by its transaction cannot be released for use by other transactions. The participant can make 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 17.5. If 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.

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 17.5 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 necessary 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 the time a timeout period on a lock expires. As no decision has been made at this stage, the participant can decide to *abort* unilaterally.

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, the participants and the communications 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 preceding section, 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, but they are more expensive in terms of 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 17.2 and Bernstein *et al*. [1987].

Figure 17.7 Operations in coordinator for nested transactions

openSubTransaction(trans) → *subTrans*
Opens a new subtransaction whose parent is *trans* and returns a unique subtransaction identifier.

getStatus(trans)→ *committed, aborted, provisional*
Asks the coordinator to report on the status of the transaction *trans*. Returns values representing one of the following: *committed*, *aborted* or *provisional*.

17.3.2 Two-phase commit protocol for nested transactions

The outermost transaction in a set of nested transactions is called the *top-level transaction*. Transactions other than the top-level transaction are called *subtransactions*. In Figure 17.1(b), T is the top-level transaction and T_1, T_2, T_{11}, T_{12}, T_{21} and T_{22} are subtransactions. T_1 and T_2 are child transactions of T, which is referred to as their *parent*. Similarly, T_{11} and T_{12} are child transactions of T_1, and T_{21} and T_{22} are child transactions of T_2. Each subtransaction starts after its parent and finishes before it. Thus, for example, T_{11} and T_{12} start after T_1 and finish before it.

When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. A provisional commit is different from being prepared to commit: nothing is backed up in permanent storage. If the server crashes subsequently, its replacement will not be able to commit. After all subtransactions have completed, the provisionally committed ones participate in a two-phase commit protocol, in which servers of provisionally committed subtransactions express their intention to commit and those with an aborted ancestor will abort. Being prepared to commit guarantees a subtransaction will be able to commit, whereas a provisional commit only means it has finished correctly – and will probably agree to commit when it is subsequently asked to.

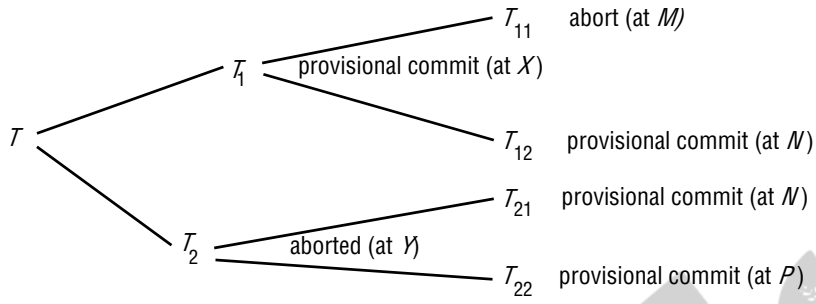
A coordinator for a subtransaction will provide an operation to open a subtransaction, together with an operation enabling that coordinator to enquire whether its parent has yet committed or aborted, as shown in Figure 17.7.

A client starts a set of nested transactions by opening a top-level transaction with an *openTransaction* operation, which returns a transaction identifier for the top-level transaction. The client starts a subtransaction by invoking the *openSubTransaction* operation, whose argument specifies its parent transaction. The new subtransaction automatically *joins* the parent transaction, and a transaction identifier for a subtransaction is returned.

An identifier for a subtransaction must be an extension of its parent's TID, constructed in such a way that the identifier of the parent or top-level transaction of a subtransaction can be determined from its own transaction identifier. In addition, all subtransaction identifiers should be globally unique. The client makes a set of nested transactions come to completion by invoking *closeTransaction* or *abortTransaction* on the coordinator of the top-level transaction.

Meanwhile, each of the nested transactions carries out its operations. When they are finished, the server managing a subtransaction records information as to whether the

Figure 17.8 Transaction *T* decides whether to commit



subtransaction committed provisionally or aborted. Note that if its parent aborts, then the subtransaction will be forced to abort too.

Recall from Chapter 16 that a parent transaction – including a top-level transaction – can commit even if one of its child subtransactions has aborted. In such cases, the parent transaction will be programmed to take different actions according to whether a subtransaction has committed or aborted. For example, consider a banking transaction that is designed to perform all the 'standing orders' at a branch on a particular day. This transaction is expressed as several nested *Transfer* subtransactions, each of which consists of nested *deposit* and *withdraw* subtransactions. We assume that when an account is overdrawn, *withdraw* aborts and then the corresponding *Transfer* aborts. But there is no need to abort all the standing orders just because one *Transfer* subtransaction aborts. Instead of aborting, the top-level transaction will note the *Transfer* subtransactions that aborted and take appropriate actions.

Consider the top-level transaction *T* and its subtransactions shown in Figure 17.8, which is based on Figure 17.1(b). Each subtransaction has either provisionally committed or aborted. For example, T_{12} has provisionally committed and T_{11} has aborted, but the fate of T_{12} depends on its parent T_1 and eventually on the top-level transaction, *T*. Although T_{21} and T_{22} have both provisionally committed, T_2 has aborted and this means that T_{21} and T_{22} must also abort. Suppose that *T* decides to commit in spite of the fact that T_2 has aborted, and that T_1 decides to commit in spite of the fact that T_{11} has aborted.

When a top-level transaction completes, its coordinator carries out a two-phase commit protocol. The only reason for a participant subtransaction being unable to complete is if it has crashed since it completed its provisional commit. Recall that when each subtransaction was created, it *joined* its parent transaction. Therefore, the coordinator of each parent transaction has a list of its child subtransactions. When a nested transaction provisionally commits, it reports its status and the status of its descendants to its parent. When a nested transaction aborts, it just reports *abort* to its parent without giving any information about its descendants. Eventually, the top-level transaction receives a list of all the subtransactions in the tree, together with the status of each. Descendants of aborted subtransactions are omitted from this list.

Figure 17.9 Information held by coordinators of nested transactions

Coordinator of transaction	*Child transactions*	*Participant*	*Provisional commit list*	*Abort list*
T	T_1, T_2	yes	T_1, T_{12}	T_{11}, T_2
T_1	T_{11}, T_{12}	yes	T_1, T_{12}	T_{11}
T_2	T_{21}, T_{22}	no (aborted)		T_2
T_{11}		no (aborted)		T_{11}
T_{12}, T_{21}		T_{12} but not T_{21}*	T_{21}, T_{12}	
T_{22}		no (parent aborted)	T_{22}	

* T_{21}'s parent has aborted

The information held by each coordinator in the example in Figure 17.8 is shown in Figure 17.9. Note that T_{12} and T_{21} share a coordinator as they both run at server N. When subtransaction T_2 aborted, it reported the fact to its parent, T, but without passing on any information about its subtransactions T_{21} and T_{22}. A subtransaction is an *orphan* if one of its ancestors aborts, either explicitly or because its coordinator crashes. In our example, subtransactions T_{21} and T_{22} are orphans because their parent aborted without passing information about them to the top-level transaction. Their coordinator can, however, make enquiries about the status of their parent by using the *getStatus* operation. A provisionally committed subtransaction of an aborted transaction should be aborted, irrespective of whether the top-level transaction eventually commits.

The top-level transaction plays the role of coordinator in the two-phase commit protocol, and the participant list consists of the coordinators of all the subtransactions in the tree that have provisionally committed but do not have aborted ancestors. By this stage, the logic of the program has determined that the top-level transaction should try to commit whatever is left, in spite of some aborted subtransactions. In Figure 17.8, the coordinators of T, T_1 and T_{12} are participants and will be asked to vote on the outcome. If they vote to commit, then they must *prepare* their transactions by saving the state of the objects in permanent storage. This state is recorded as belonging to the top-level transaction of which it will form a part. The two-phase commit protocol may be performed in either a hierarchic manner or a flat manner.

The second phase of the two-phase commit protocol is the same as for the non-nested case. The coordinator collects the votes and then informs the participants as to the outcome. When it is complete, coordinator and participants will have committed or aborted their transactions.

Hierarchic two-phase commit protocol • In this approach, the two-phase commit protocol becomes a multi-level nested protocol. The coordinator of the top-level transaction communicates with the coordinators of the subtransactions for which it is the immediate parent. It sends *canCommit?* messages to each of the latter, which in turn pass them on to the coordinators of their child transactions (and so on down the tree). Each participant collects the replies from its descendants before replying to its parent. In our example, T sends *canCommit?* messages to the coordinator of T_1 and then T_1 sends *canCommit?* messages to T_{12} asking about descendants of T_1. The protocol does not include the coordinators of transactions such as T_2, which has aborted. Figure 17.10

Figure 17.10 *canCommit?* for hierarchic two-phase commit protocol

canCommit?(trans, subTrans) → *Yes / No*
Call from coordinator to coordinator of child subtransaction to ask whether it can commit a subtransaction *subTrans*. The first argument, *trans*, is the transaction identifier of the top-level transaction. Participant replies with its vote, *Yes / No*.

shows the arguments required for *canCommit?*. The first argument is the TID of the top-level transaction, for use when preparing the data. The second argument is the TID of the participant making the *canCommit?* call. The participant receiving the call looks in its transaction list for any provisionally committed transaction or subtransaction matching the TID in the second argument. For example, the coordinator of T_{12} is also the coordinator of T_{21}, since they run in the same server, but when it receives the *canCommit?* call, the second argument will be T_1 and it will deal only with T_{12}.

If a participant finds any subtransactions that match the second argument, it prepares the objects and replies with a *Yes* vote. If it fails to find any, then it must have crashed since it performed the subtransaction and it replies with a *No* vote.

Flat two-phase commit protocol • In this approach, the coordinator of the top-level transaction sends *canCommit?* messages to the coordinators of all of the subtransactions in the provisional commit list – in our example, to the coordinators of T_1 and T_{12}. During the commit protocol, the participants refer to the transaction by its top-level TID. Each participant looks in its transaction list for any transaction or subtransaction matching that TID. For example, the coordinator of T_{12} is also the coordinator of T_{21}, since they run in the same server (*N*).

Unfortunately, this does not provide sufficient information to enable correct actions by participants such as the coordinator at server *N* that have a mix of provisionally committed and aborted subtransactions. If *N*'s coordinator is just asked to commit *T* it will end up by committing both T_{12} and T_{21}, because, according to its local information, both have provisionally committed. This is wrong in the case of T_{21}, because its parent, T_2, has aborted. To allow for such cases, the *canCommit?* operation for the flat commit protocol has a second argument that provides a list of aborted subtransactions, as shown in Figure 17.11. A participant can commit descendants of the top-level transaction unless they have aborted ancestors. When a participant receives a *canCommit?* request, it does the following:

- If the participant has any provisionally committed transactions that are descendants of the top-level transaction, *trans*, it:
  - checks that they do not have aborted ancestors in the *abortList*, then prepares to commit (by recording the transaction and its objects in permanent storage);

Figure 17.11 *canCommit?* for flat two-phase commit protocol

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

  - aborts those with aborted ancestors;
  - sends a *Yes* vote to the coordinator.
- If the participant does not have a provisionally committed descendent of the top-level transaction, it must have failed since it performed the subtransaction and it sends a *No* vote to the coordinator.

A comparison of the two approaches • The hierarchic protocol has the advantage that at each stage, the participant only need look for subtransactions of its immediate parent, whereas the flat protocol needs to have the abort list in order to eliminate transactions whose parents have aborted. Moss [1985] preferred the flat algorithm because it allows the coordinator of the top-level transaction to communicate directly with all of the participants, whereas the hierarchic variant involves passing a series of messages down and up the tree in stages.

Timeout actions • The two-phase commit protocol for nested transactions can cause the coordinator or a participant to be delayed at the same three steps as in the non-nested version. There is a fourth step at which subtransactions can be delayed. Consider provisionally committed child subtransactions of aborted subtransactions: they do not necessarily get informed of the outcome of the transaction. In our example, T_{22} is such a subtransaction – it has provisionally committed, but as its parent T_2 has aborted, it does not become a participant. To deal with such situations, any subtransaction that has not received a *canCommit?* message will make an enquiry after a timeout period. The *getStatus* operation in Figure 17.7 allows a subtransaction to enquire whether its parent has committed or aborted. To make such enquiries possible, the coordinators of aborted subtransactions need to survive for a period. If an orphaned subtransaction cannot contact its parent, it will eventually abort.

17.4 Concurrency control in distributed transactions

Each server manages a set of objects and is responsible for ensuring that they remain consistent when accessed by concurrent transactions. Therefore, each server is responsible for applying concurrency control to its own objects. The members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner.

This implies that if transaction T is before transaction U in their conflicting access to objects at one of the servers, then they must be in that order at all of the servers whose objects are accessed in a conflicting manner by both T and U.

17.4.1 Locking

In a distributed transaction, the locks on an object are held locally (in the same server). The local lock manager can decide whether to grant a lock or make the requesting transaction wait. However, it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction. When locking is used for concurrency control, the objects remain locked and are unavailable

for other transactions during the atomic commit protocol, although an aborted transaction releases its locks after phase 1 of the protocol.

As lock managers in different servers set their locks independently of one another, it is possible that different servers may impose different orderings on transactions. Consider the following interleaving of transactions *T* and *U* at servers *X* and *Y*:

T			*U*		
write(A)	at *X*	locks *A*			
			write(B)	at *Y*	locks *B*
read(B)	at *Y*	waits for *U*			
			read(A)	at *X*	waits for *T*

The transaction *T* locks object *A* at server *X*, and then transaction *U* locks object *B* at server *Y*. After that, *T* tries to access *B* at server *Y* and waits for *U*'s lock. Similarly, transaction *U* tries to access *A* at server *X* and has to wait for *T*'s lock. Therefore, we have *T* before *U* in one server and *U* before *T* in the other. These different orderings can lead to cyclic dependencies between transactions, giving rise to a distributed deadlock situation. The detection and resolution of distributed deadlocks is discussed in Section 17.5. When a deadlock is detected, a transaction is aborted to resolve the deadlock. In this case, the coordinator will be informed and will abort the transaction at the participants involved in the transaction.

17.4.2 Timestamp ordering concurrency control

In a single server transaction, the coordinator issues a unique timestamp to each transaction when it starts. Serial equivalence is enforced by committing the versions of objects in the order of the timestamps of transactions that accessed them. In distributed transactions, we require that each coordinator issue globally unique timestamps. A globally unique transaction timestamp is issued to the client by the first coordinator accessed by a transaction. The transaction timestamp is passed to the coordinator at each server whose objects perform an operation in the transaction.

The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. For example, if the version of an object accessed by transaction *U* commits after the version accessed by *T* at one server, if *T* and *U* access the same object as one another at other servers they must commit them in the same order. To achieve the same ordering at all the servers, the coordinators must agree as to the ordering of their timestamps. A timestamp consists of a <*local timestamp, server-id*> pair. The agreed ordering of pairs of timestamps is based on a comparison in which the *server-id* part is less significant.

The same ordering of transactions can be achieved at all the servers even if their local clocks are not synchronized. However, for reasons of efficiency it is required that the timestamps issued by one coordinator be roughly synchronized with those issued by the other coordinators. When this is the case, the ordering of transactions generally

corresponds to the order in which they are started in real time. Timestamps can be kept roughly synchronized by the use of synchronized local physical clocks (see Chapter 14).

When timestamp ordering is used for concurrency control, conflicts are resolved as each operation is performed using the rules given in Section 16.6. If the resolution of a conflict requires a transaction to be aborted, the coordinator will be informed and it will abort the transaction at all the participants. Therefore any transaction that reaches the client request to commit should always be able to commit, and participants in the two-phase commit protocol will normally agree to commit. The only situation in which a participant will not agree to commit is if it has crashed during the transaction.

17.4.3 Optimistic concurrency control

Recall that with optimistic concurrency control, each transaction is validated before it is allowed to commit. Transaction numbers are assigned at the start of validation and transactions are serialized according to the order of the transaction numbers. A distributed transaction is validated by a collection of independent servers, each of which validates transactions that access its own objects. This validation takes place during the first phase of the two-phase commit protocol.

Consider the following interleavings of transactions *T* and *U*, which access objects *A* and *B* at servers *X* and *Y*, respectively:

T		*U*	
read(A)	at *X*	*read(B)*	at *Y*
write(A)		*write(B)*	
read(B)	at *Y*	*read(A)*	at *X*
write(B)		*write(A)*	

The transactions access the objects in the order *T* before *U* at server *X* and in the order *U* before *T* at server *Y*. Now suppose that *T* and *U* start validation at about the same time, but server *X* validates *T* first and server *Y* validates *U* first. Recall that Section 16.5 recommends a simplification of the validation protocol that makes a rule that only one transaction may perform validation and update phases at a time. Therefore each server will be unable to validate the other transaction until the first one has completed. This is an example of commitment deadlock.

The validation rules in Section 16.5 assume that validation is fast, which is true for single-server transactions. However, in a distributed transaction, the two-phase commit protocol may take some time to complete, and other transactions will be prevented from entering validation until a decision on the current transaction has been obtained. In distributed optimistic transactions, each server applies a parallel validation protocol. This is an extension of either backward or forward validation to allow multiple transactions to be in the validation phase at the same time. In this extension, rule 3 must be checked as well as rule 2 for backward validation. That is, the write set of the transaction being validated must be checked for overlaps with the write set of earlier overlapping transactions. Kung and Robinson [1981] describe parallel validation.

If parallel validation is used, transactions will not suffer from commitment deadlock. However, if servers simply perform independent validations, it is possible that different servers in a distributed transaction may serialize the same set of transactions in different orders – for example, with *T* before *U* at server *X* and *U* before *T* at server *Y*, in our example.

The servers of distributed transactions must prevent this happening. One approach is that after a local validation by each server, a global validation is carried out [Ceri and Owicki 1982]. The global validation checks that the combination of the orderings at the individual servers is serializable; that is, that the transaction being validated is not involved in a cycle.

Another approach is that all of the servers of a particular transaction use the same globally unique transaction number at the start of the validation [Schlageter 1982]. The coordinator of the two-phase commit protocol is responsible for generating the globally unique transaction number and passes it to the participants in the *canCommit?* messages. As different servers may coordinate different transactions, the servers must (as in the distributed timestamp ordering protocol) have an agreed order for the transaction numbers they generate.

Agrawal *et al.* [1987] have proposed a variation of Kung and Robinson's algorithm that favours read-only transactions, together with an algorithm called MVGV (multi-version generalized validation). MVGV is a form of parallel validation that ensures that transaction numbers reflect serial order, but it requires that in some cases, other transactions are unable to read their effects immediately after they have committed. It also allows the transaction number to be changed so as to permit some transactions to validate that otherwise would have failed. The paper also proposes an algorithm for committing distributed transactions. It is similar to Schlageter's proposal in that a global transaction number has to be found. At the end of the read phase, the coordinator proposes a value for the global transaction number and each participant attempts to validate its local transactions using that number. However, if the proposed global transaction number is too small, some participants may not be able to validate their transactions, and they will have to negotiate with the coordinator for an increased number. If no suitable number can be found, then those participants will have to abort their transactions. Eventually, if all of the participants can validate their transactions, the coordinator will have received proposals for transaction numbers from each of them. If common numbers can be found then the transaction will be committed.

17.5 Distributed deadlocks

The discussion of deadlocks in Section 16.4 showed that deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks. Using timeouts to resolve possible deadlocks is a clumsy approach – it is difficult to choose an appropriate timeout interval, and transactions may be aborted unnecessarily. With deadlock detection schemes, a transaction is aborted only when it is involved in a deadlock. Most deadlock detection schemes operate by finding cycles in the transaction wait-for graph. In a distributed system involving multiple servers being accessed by multiple transactions, a global

Figure 17.12 Interleavings of transactions *U*, *V* and *W*

U		*V*		*W*	
d.deposit(10)	lock *D*				
		b.deposit(10)	lock *B*		
a.deposit(20)	lock *A*		at *Y*		
	at *X*				
				c.deposit(30)	lock *C*
b.withdraw(30)	wait at *Y*				at *Z*
		c.withdraw(20)	wait at *Z*		
				a.withdraw(20)	wait at *X*

wait-for graph can in theory be constructed from the local ones. There can be a cycle in the global wait-for graph that is not in any single local one – that is, there can be a *distributed deadlock*. Recall that the wait-for graph is a directed graph in which nodes represent transactions and objects, and edges represent either an object held by a transaction or a transaction waiting for an object. There is a deadlock if and only if there is a cycle in the wait-for graph.

Figure 17.12 shows the interleavings of the transactions *U*, *V* and *W* involving the objects *A* and *B* managed by servers *X* and *Y* and objects *C* and *D* managed by server *Z*.

The complete wait-for graph in Figure 17.13(a) shows that a deadlock cycle consists of alternate edges, which represent a transaction waiting for an object and an object held by a transaction. As any transaction can only be waiting for one object at a time, objects can be left out of wait-for graphs, as shown in Figure 17.13(b).

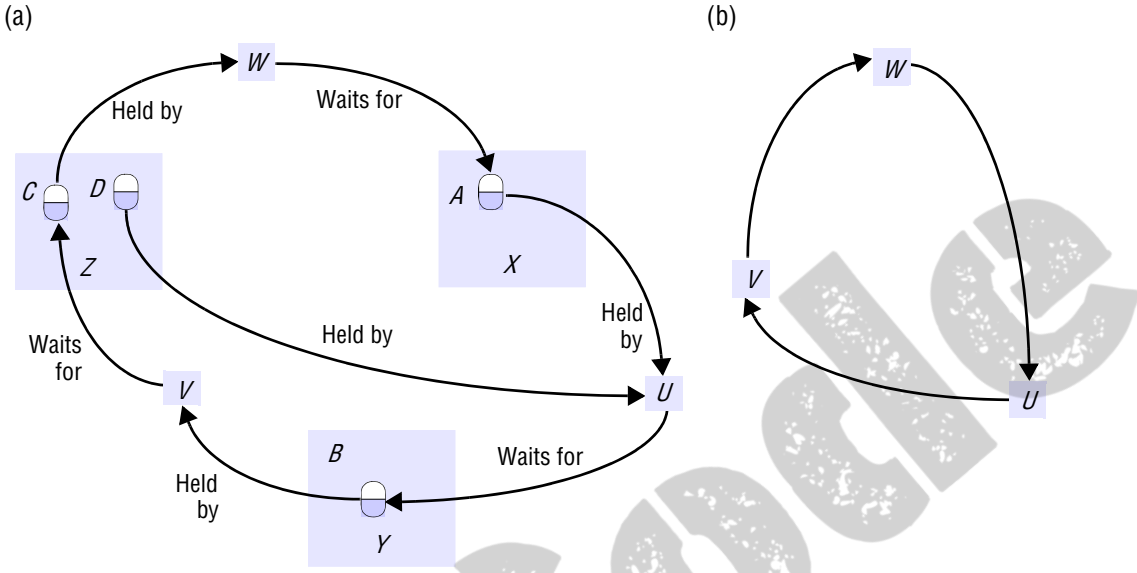
Detection of a distributed deadlock requires a cycle to be found in the global transaction wait-for graph that is distributed among the servers that were involved in the transactions. Local wait-for graphs can be built by the lock manager at each server, as discussed in Chapter 16. In the above example, the local wait-for graphs of the servers are:

server *Y*: $U \rightarrow V$ (added when *U* requests *b.withdraw(30)*)
server *Z*: $V \rightarrow W$ (added when *V* requests *c.withdraw(20)*)
server *X*: $W \rightarrow U$ (added when *W* requests *a.withdraw(20)*)

As the global wait-for graph is held in part by each of the several servers involved, communication between these servers is required to find cycles in the graph.

A simple solution is to use centralized deadlock detection, in which one server takes on the role of global deadlock detector. From time to time, each server sends the latest copy of its local wait-for graph to the global deadlock detector, which amalgamates the information in the local graphs in order to construct a global wait-for graph. The global deadlock detector checks for cycles in the global wait-for graph.

Figure 17.13 Distributed deadlock



When it finds a cycle, it makes a decision on how to resolve the deadlock and tells the servers which transaction to abort.

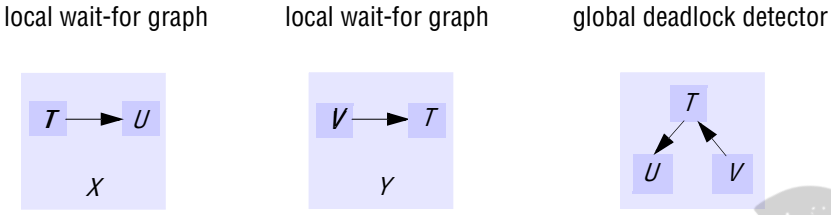
Centralized deadlock detection is not a good idea, because it depends on a single server to carry it out. It suffers from the usual problems associated with centralized solutions in distributed systems – poor availability, lack of fault tolerance and no ability to scale. In addition, the cost of the frequent transmission of local wait-for graphs is high. If the global graph is collected less frequently, deadlocks may take longer to be detected.

Phantom deadlocks • A deadlock that is 'detected' but is not really a deadlock is called a *phantom deadlock*. In distributed deadlock detection, information about wait-for relationships between transactions is transmitted from one server to another. If there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected. As this procedure will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

Consider the case of a global deadlock detector that receives local wait-for graphs from servers X and Y, as shown in Figure 17.14. Suppose that transaction U then releases an object at server X and requests the one held by V at server Y. Suppose also that the global detector receives server Y's local graph before server X's. In this case, it would detect a cycle $T \rightarrow U \rightarrow V \rightarrow T$, although the edge $T \rightarrow U$ no longer exists. This is an example of a phantom deadlock.

The observant reader will have realized that if transactions are using two-phase locks, they cannot release objects and then obtain more objects, and phantom deadlock

Figure 17.14 Local and global wait-for graphs



cycles cannot occur in the way suggested above. Consider the situation in which a cycle $T \rightarrow U \rightarrow V \rightarrow T$ is detected: either this represents a deadlock or each of the transactions T, U and V must eventually commit. It is actually impossible for any of them to commit, because each of them is waiting for an object that will never be released.

A phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure. For example, if there is a cycle $T \rightarrow U \rightarrow V \rightarrow T$ and U aborts after the information concerning U has been collected, then the cycle has been broken already and there is no deadlock.

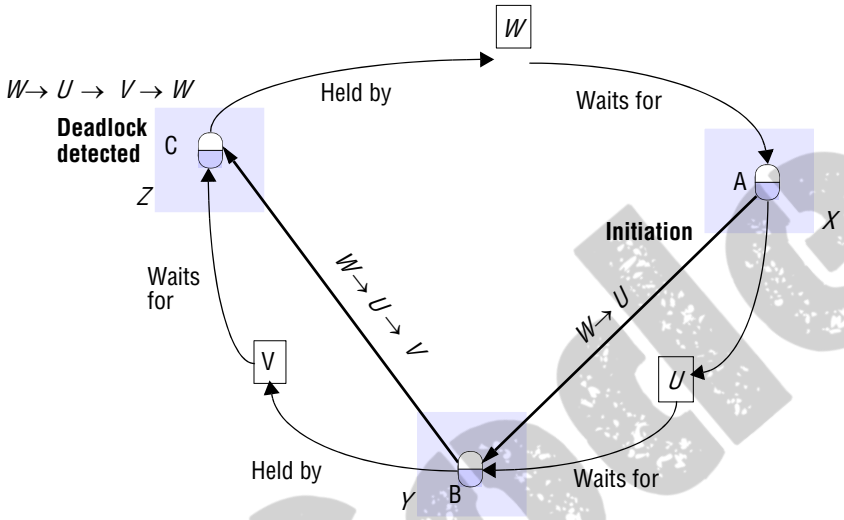
Edge chasing • A distributed approach to deadlock detection uses a technique called *edge chasing* or *path pushing*. In this approach, the global wait-for graph is not constructed, but each of the servers involved has knowledge about some of its edges. The servers attempt to find cycles by forwarding messages called *probes*, which follow the edges of the graph throughout the distributed system. A probe message consists of transaction wait-for relationships representing a path in the global wait-for graph.

The question is, when should a server send out a probe? Consider the situation at server X in Figure 17.13. This server has just added the edge $W \rightarrow U$ to its local wait-for graph, and at this time transaction U is waiting to access object B, which transaction V holds at server Y. This edge could possibly be part of a cycle such as $V \rightarrow T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow W \rightarrow U \rightarrow V$ involving transactions using objects at other servers. This indicates that there is a potential distributed deadlock cycle, which could be found by sending out a probe to server Y.

Now consider the situation a little earlier, when server Z added the edge $V \rightarrow W$ to its local graph. At this point in time W is not waiting, so there is no point in sending out a probe.

Each distributed transaction starts at a server (called the coordinator of the transaction) and moves to several other servers (called participants in the transaction), which can communicate with the coordinator. At any point in time, a transaction can be either active or waiting at just one of these servers. The coordinator is responsible for recording whether the transaction is active or is waiting for a particular object, and participants can get this information from their coordinator. Lock managers inform coordinators when transactions start waiting for objects and when transactions acquire objects and become active again. When a transaction is aborted to break a deadlock, its coordinator will inform the participants and all of its locks will be removed, with the effect that all edges involving that transaction will be removed from the local wait-for graphs.

Figure 17.15 Probes transmitted to detect deadlock



Edge-chasing algorithms have three steps:

Initiation: When a server notes that a transaction T starts waiting for another transaction U, where U is waiting to access an object at another server, it initiates detection by sending a probe containing the edge $< T \rightarrow U >$ to the server of the object at which transaction U is blocked. If U is sharing a lock, probes are sent to all the holders of the lock. Sometimes further transactions may start sharing the lock later on, in which case probes can be sent to them too.

Detection: Detection consists of receiving probes and deciding whether a deadlock has occurred and whether to forward the probes.

For example, when a server of an object receives a probe $< T \rightarrow U >$ (indicating that T is waiting for a transaction U that holds a local object), it checks to see whether U is also waiting. If it is, the transaction it waits for (for example, V) is added to the probe (making it $< T \rightarrow U \rightarrow V >$), and if the new transaction (V) is waiting for another object elsewhere, the probe is forwarded.

In this way, paths through the global wait-for graph are built one edge at a time. Before forwarding a probe, the server checks to see whether the transaction (for example, T) it has just added has caused the probe to contain a cycle (for example, $< T \rightarrow U \rightarrow V \rightarrow T >$). If this is the case, it has found a cycle in the graph and a deadlock has been detected.

Resolution: When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.

In our example, the following steps describe how deadlock detection is initiated and the probes that are forwarded during the corresponding detection phase:

- Server X initiates detection by sending probe $< W \rightarrow U >$ to the server of B (Server Y).
- Server Y receives probe $< W \rightarrow U >$, notes that B is held by V and appends V to the probe to produce $< W \rightarrow U \rightarrow V >$. It notes that V is waiting for C at server Z. This probe is forwarded to server Z.
- Server Z receives probe $< W \rightarrow U \rightarrow V >$, notes C is held by W and appends W to the probe to produce $< W \rightarrow U \rightarrow V \rightarrow W >$.

This path contains a cycle. The server detects a deadlock. One of the transactions in the cycle must be aborted to break the deadlock. The transaction to be aborted can be chosen according to transaction priorities, which are described shortly.

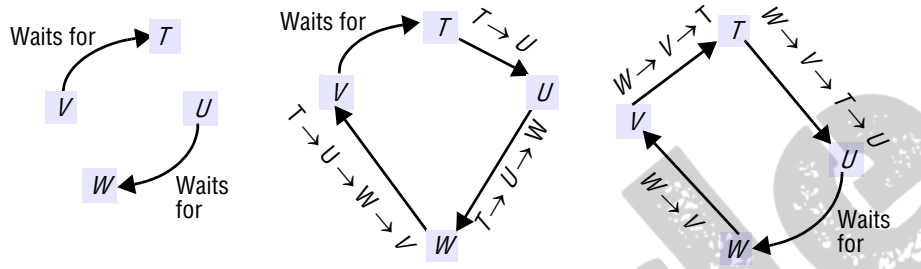
Figure 17.15 shows the progress of the probe messages from the initiation by the server of A to the deadlock detection by the server of C. Probes are shown as heavy arrows, objects as ovals (as usual) and transaction coordinators as rectangles. Each probe is shown as going directly from one object to another. In reality, before a server transmits a probe to another server, it consults the coordinator of the last transaction in the path to find out whether the latter is waiting for another object elsewhere. For example, before the server of B transmits the probe $<W \rightarrow U \rightarrow V>$ it consults the coordinator of V to find out that V is waiting for C. In most of the edge-chasing algorithms, the servers of objects send probes to transaction coordinators, which then forward them (if the transaction is waiting) to the server of the object for which the transaction is waiting. In our example, the server of B transmits the probe $<W \rightarrow U \rightarrow V>$ to the coordinator of V, which then forwards it to the server of C. This shows that when a probe is forwarded, two messages are required.

The above algorithm should find any deadlock that occurs, provided that waiting transactions do not abort and there are no failures such as lost messages or servers crashing. To understand this, consider a deadlock cycle in which the last transaction, W, starts waiting and completes the cycle. When W starts waiting for an object, the server initiates a probe that goes to the server of the object held by each transaction for which W is waiting. The recipients extend and forward the probes to the servers of objects requested by all waiting transactions they find. Thus every transaction that W waits for, directly or indirectly, will be added to the probe unless a deadlock is detected. When there is a deadlock, W is waiting for itself indirectly. Therefore, the probe will return to the object that W holds.

It might appear that large numbers of messages are sent in order to detect deadlocks. In the above example, we see two probe messages to detect a cycle involving three transactions. Each of the probe messages is in general two messages (from object to coordinator and then from coordinator to object).

A probe that detects a cycle involving N transactions will be forwarded by $(N - 1)$ transaction coordinators via $(N - 1)$ servers of objects, requiring $2(N - 1)$ messages. Fortunately, the majority of deadlocks involve cycles containing only two transactions, and there is no need for undue concern about the number of messages involved. This observation has been made from studies of databases. It can also be argued by considering the probability of conflicting access to objects (see Bernstein *et al*. [1987]).

Figure 17.16 Two probes initiated



Transaction priorities • In the above algorithm, every transaction involved in a deadlock cycle can cause deadlock detection to be initiated. The effect of several transactions in a cycle initiating deadlock detection is that detection may happen at several different servers in the cycle, with the result that more than one transaction in the cycle is aborted.

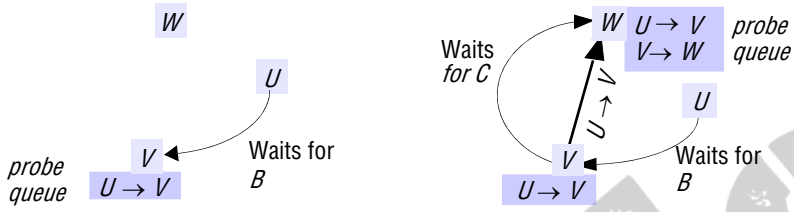
In Figure 17.16(a), consider transactions T, U, V and W, where U is waiting for W and V is waiting for T. At about the same time, T requests the object held by U and W requests the object held by V. Two separate probes, $< T \rightarrow U >$ and $< W \rightarrow V >$, are initiated by the servers of these objects and are circulated until deadlocks are detected by each of the servers. In Figure 17.16(b), the cycle is $< T \rightarrow U \rightarrow W \rightarrow V \rightarrow T >$, and in Figure 17.16 (c), the cycle is $< W \rightarrow V \rightarrow T \rightarrow U \rightarrow W >$.

In order to ensure that only one transaction in a cycle is aborted, transactions are given *priorities* in such a way that all transactions are totally ordered. Timestamps, for example, may be used as priorities. When a deadlock cycle is found, the transaction with the lowest priority is aborted. Even if several different servers detect the same cycle, they will all reach the same decision as to which transaction is to be aborted. We write $T > U$ to indicate that T has higher priority than U. In the above example, assume $T > U > V > W$. Then the transaction W will be aborted when either of the cycles $< T \rightarrow U \rightarrow W \rightarrow V \rightarrow T >$ or $< W \rightarrow V \rightarrow T \rightarrow U \rightarrow W >$ is detected.

It might appear that transaction priorities could also be used to reduce the number of situations that cause deadlock detection to be initiated, by using the rule that detection is initiated only when a higher-priority transaction starts to wait for a lower-priority one. In our example in Figure 17.16, as $T > U$ the initiating probe $< T \rightarrow U >$ would be sent, but as $W < V$ the initiating probe $< W \rightarrow V >$ would not be sent. If we assume that when a transaction starts waiting for another transaction it is equally likely that the waiting transaction has higher or lower priority than the waited-for transaction, then the use of this rule is likely to reduce the number of probe messages by about half.

Transaction priorities could also be used to reduce the number of probes that are forwarded. The general idea is that probes should travel 'downhill' – that is, from transactions with higher priorities to transactions with lower priorities. To this end, servers use the rule that they do not forward any probe to a holder that has higher priority than the initiator. The argument for doing this is that if the holder is waiting for another transaction, it must have initiated detection by sending a probe when it started waiting.

Figure 17.17 Probes travel downhill



However, there is a pitfall associated with these apparent improvements. In our example in Figure 17.15 transactions U, V and W are executed in an order in which U is waiting for V and V is waiting for W when W starts waiting for U. Without priority rules, detection is initiated when W starts waiting by sending a probe $< W \rightarrow U >$. Under the priority rule, this probe will not be sent because $W < U$ and the deadlock will not be detected.

The problem is that the order in which transactions start waiting can determine whether or not a deadlock will be detected. The above pitfall can be avoided by using a scheme in which coordinators save copies of all the probes received on behalf of each transaction in a *probe queue*. When a transaction starts waiting for an object, it forwards the probes in its queue to the server of the object, which propagates the probes on downhill routes.

In our example in Figure 17.15, when U starts waiting for V, the coordinator of V will save the probe $< U \rightarrow V >$ – see Figure 17.17(a). Then when V starts waiting for W, the coordinator of W will store $< V \rightarrow W >$ and V will forward its probe queue, $< U \rightarrow V >$, to W. (See Figure 17.17(b), in which W's probe queue has $< U \rightarrow V >$ and $< V \rightarrow W >$.) When W starts waiting for A it will forward its probe queue, $< U \rightarrow V \rightarrow W >$, to the server of A, which notes the new dependency $W \rightarrow U$ and combines it with the information in the probe received to determine that $U \rightarrow V \rightarrow W \rightarrow U$. Deadlock is detected.

When an algorithm requires probes to be stored in probe queues, it also requires arrangements to pass on probes to new holders and to discard probes that refer to transactions that have been committed or aborted. If relevant probes are discarded, undetected deadlocks may occur, and if outdated probes are retained, false deadlocks may be detected. This adds much to the complexity of any edge-chasing algorithm. Readers who are interested in the details of such algorithms should see Sinha and Natarajan [1985] and Choudhary *et al*. [1989], who present algorithms for use with exclusive locks. But they will see that Choudhary *et al*. showed that Sinha and Natarajan's algorithm is incorrect: it fails to detect all deadlocks and may even report false deadlocks. Kshemkalyani and Singhal [1991] corrected the algorithm of Choudhary *et al*. and provided a proof of correctness for the corrected algorithm. In a subsequent paper, Kshemkalyani and Singhal [1994] argued that distributed deadlocks are not very well understood because there is no global state or time in a distributed

system. In fact, any cycle that has been collected may contain sections recorded at different times. In addition, sites may hear about deadlocks but may not hear that they have been resolved until after random delays. The paper describes distributed deadlocks in terms of the contents of distributed memory, using causal relationships between events at different sites.

17.6 Transaction recovery

The atomic property of transactions requires that all the effects of committed transactions and none of the effects of incomplete or aborted transactions are reflected in the objects they accessed. This property can be described in terms of two aspects: durability and failure atomicity. Durability requires that objects are saved in permanent storage and will be available indefinitely thereafter. Therefore an acknowledgement of a client's commit request implies that all the effects of the transaction have been recorded in permanent storage as well as in the server's (volatile) objects. Failure atomicity requires that effects of transactions are atomic even when the server crashes. Recovery is concerned with ensuring that a server's objects are durable and that the service provides failure atomicity.

Although file servers and database servers maintain data in permanent storage, other kinds of servers of recoverable objects need not do so except for recovery purposes. In this chapter, we assume that when a server is running it keeps all of its objects in its volatile memory and records its committed objects in a *recovery file* or files. Therefore recovery consists of restoring the server with the latest committed versions of its objects from permanent storage. Databases need to deal with large volumes of data. They generally hold the objects in stable storage on disk with a cache in volatile memory.

The requirements for durability and failure atomicity are not really independent of one another and can be dealt with by a single mechanism – the *recovery manager*. The tasks of a recovery manager are:

- to save objects in permanent storage (in a recovery file) for committed transactions;
- to restore the server's objects after a crash;
- to reorganize the recovery file to improve the performance of recovery;
- to reclaim storage space (in the recovery file).

In some cases, we require the recovery manager to be resilient to media failures. Corruption during a crash, random decay or a permanent failure can lead to failures of the recovery file, which can result in some of the data on the disk being lost. In such cases we need another copy of the recovery file. Stable storage, which is implemented so as to be very unlikely to fail by using mirrored disks or copies at a different location may be used for this purpose.

Intentions list • Any server that provides transactions needs to keep track of the objects accessed by clients' transactions. Recall from Chapter 16 that when a client opens a transaction, the server first contacted provides a new transaction identifier and

Figure 17.18 Types of entry in a recovery file

Type of entry	*Description of contents of entry*
Object	A value of an object.
Transaction status	Transaction identifier, transaction status (*prepared*, *committed*, *aborted*) and other status values used for the two-phase commit protocol.
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of <*objectID*, P_i>, where P_i is the position in the recovery file of the value of the object.

returns it to the client. Each subsequent client request within a transaction up to and including the *commit* or *abort* request includes the transaction identifier as an argument. During the progress of a transaction, the update operations are applied to a private set of tentative versions of the objects belonging to the transaction.

At each server, an *intentions list* is recorded for all of its currently active transactions – an intentions list of a particular transaction contains a list of the references and the values of all the objects that are altered by that transaction. When a transaction is committed, that transaction's intentions list is used to identify the objects it affected. The committed version of each object is replaced by the tentative version made by that transaction, and the new value is written to the server's recovery file. When a transaction aborts, the server uses the intentions list to delete all the tentative versions of objects made by that transaction.

Recall also that a distributed transaction must carry out an atomic commit protocol before it can be committed or aborted. Our discussion of recovery is based on the two-phase commit protocol, in which all the participants involved in a transaction first say whether they are prepared to commit and later, if all the participants agree, carry out the actual commit actions. If the participants cannot agree to commit, they must abort the transaction.

At the point when a participant says it is prepared to commit a transaction, its recovery manager must have saved both its intentions list for that transaction and the objects in that intentions list in its recovery file, so that it will be able to carry out the commitment later, even if it crashes in the interim.

When all the participants involved in a transaction agree to commit it, the coordinator informs the client and then sends messages to the participants to commit their part of the transaction. Once the client has been informed that a transaction has committed, the recovery files of the participating servers must contain sufficient information to ensure that the transaction is committed by all of the servers, even if some of them crash between preparing to commit and committing.

Entries in recovery file • To deal with recovery of a server that can be involved in distributed transactions, further information in addition to the values of the objects is stored in the recovery file. This information concerns the *status* of each transaction –

whether it is *committed*, *aborted* or *prepared* to commit. In addition, each object in the recovery file is associated with a particular transaction by saving the intentions list in the recovery file. Figure 17.18 shows a summary of the types of entry included in a recovery file.

The transaction status values relating to the two-phase commit protocol are discussed in Section 17.6.4. We now describe two approaches to the use of recovery files: logging and shadow versions.

17.6.1 Logging

In the logging technique, the recovery file represents a log containing the history of all the transactions performed by a server. The history consists of values of objects, transaction status entries and transaction intentions lists. The order of the entries in the log reflects the order in which transactions have prepared, committed and aborted at that server. In practice, the recovery file will contain a recent snapshot of the values of all the objects in the server followed by a history of transactions postdating the snapshot.

During the normal operation of a server, its recovery manager is called whenever a transaction prepares to commit, commits or aborts a transaction. When the server is prepared to commit a transaction, the recovery manager appends all the objects in its intentions list to the recovery file, followed by the current status of that transaction (*prepared*) together with its intentions list. When a transaction is eventually committed or aborted, the recovery manager appends the corresponding status of the transaction to its recovery file.

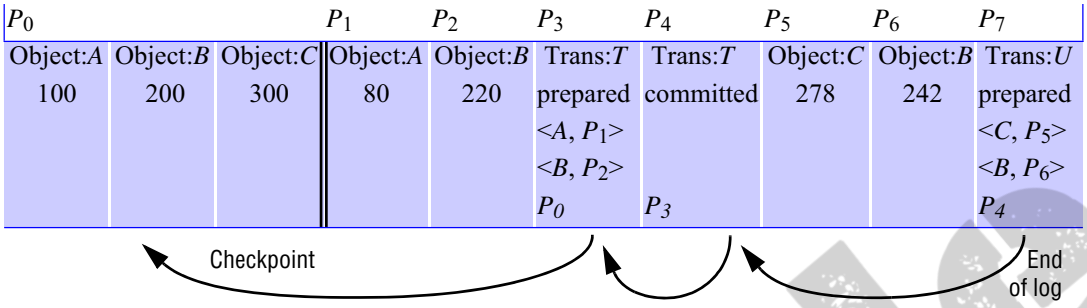
It is assumed that the append operation is atomic in the sense that it writes one or more complete entries to the recovery file. If the server fails, only the last write can be incomplete. To make efficient use of the disk, several subsequent writes can be buffered and then written to disk as a single write. An additional advantage of the logging technique is that sequential writes to disk are faster than writes to random locations.

After a crash, any transaction that does not have a *committed* status in the log is aborted. Therefore when a transaction commits, its *committed* status entry must be *forced* to the log – that is, written to the log together with any other buffered entries.

The recovery manager associates a unique identifier with each object so that the successive versions of an object in the recovery file may be associated with the server's objects. For example, a durable form of a remote object reference such as a CORBA persistent reference will do as an object identifier.

Figure 17.19 illustrates the log mechanism for the banking service transactions T and U in Figure 16.7. The log was recently reorganized, and entries to the left of the double line represent a snapshot of the values of A, B and C before transactions T and U started. In this diagram, we use the names A, B and C as unique identifiers for objects. We show the situation when transaction T has committed and transaction U has prepared but not committed. When transaction T prepares to commit, the values of objects A and B are written at positions P_1 and P_2 in the log, followed by a prepared transaction status entry for T with its intentions list ($< A, P_1 >, < B, P_2 >$). When transaction T commits, a committed transaction status entry for T is put at position P_4. Then when transaction U prepares to commit, the values of objects C and B are written at positions P_5 and P_6 in the log, followed by a prepared transaction status entry for U with its intentions list ($< C, P_5 >, < B, P_6 >$).

Figure 17.19 Log for banking service



Each transaction status entry contains a pointer to the position in the recovery file of the previous transaction status entry to enable the recovery manager to follow the transaction status entries in reverse order through the recovery file. The last pointer in the sequence of transaction status entries points to the checkpoint.

Recovery of objects • When a server is replaced after a crash, it first sets default initial values for its objects and then hands over to its recovery manager. The recovery manager is responsible for restoring the server's objects so that they include all the effects of the committed transactions performed in the correct order and none of the effects of incomplete or aborted transactions.

The most recent information about transactions is at the end of the log. There are two approaches to restoring the data from the recovery file. In the first, the recovery manager starts at the beginning and restores the values of all of the objects from the most recent checkpoint (discussed in the next section). It then reads in the values of each of the objects, associates them with their transaction's intentions lists and for committed transactions replaces the values of the objects. In this approach, the transactions are replayed in the order in which they were executed and there could be a large number of them. In the second approach, the recovery manager will restore a server's objects by 'reading the recovery file backwards'. The recovery file has been structured so that there is a backwards pointer from each transaction status entry to the next. The recovery manager uses transactions with *committed* status to restore those objects that have not yet been restored. It continues until it has restored all of the server's objects. This has the advantage that each object is restored once only.

To recover the effects of a transaction, a recovery manager gets the corresponding intentions list from its recovery file. The intentions list contains the identifiers and positions in the recovery file of values of all the objects affected by the transaction.

If the server fails at the point reached in Figure 17.19, its recovery manager will recover the objects as follows. It starts at the last transaction status entry in the log (at P_7) and concludes that transaction U has not committed and its effects should be ignored. It then moves to the previous transaction status entry in the log (at P_4) and concludes that transaction T has committed. To recover the objects affected by transaction T, it moves to the previous transaction status entry in the log (at P_3) and finds

the intentions list for T (< A, P_1 >, < B, P_2 >). It then restores objects A and B from the values at P_1 and P_2. As it has not yet restored C, it moves back to P_0, which is a checkpoint, and restores C.

To help with subsequent reorganization of the recovery file, the recovery manager notes all the prepared transactions it finds during the process of restoring the server's objects. For each prepared transaction, it adds an aborted transaction status to the recovery file. This ensures that in the recovery file, every transaction is eventually shown as either committed or aborted.

The server could fail again during the recovery procedures. It is essential that recovery be idempotent, in the sense that it can be done any number of times with the same effect. This is straightforward under our assumption that all the objects are restored to volatile memory. In the case of a database, which keeps its objects in permanent storage with a cache in volatile memory, some of the objects in permanent storage will be out of date when a server is replaced after a crash. Therefore the recovery manager has to restore the objects in permanent storage. If it fails during recovery, the partially restored objects will still be there. This makes idempotence a little harder to achieve.

Reorganizing the recovery file • A recovery manager is responsible for reorganizing its recovery file so as to make the process of recovery faster and to reduce its use of space. If the recovery file is never reorganized, then the recovery process must search backwards through the recovery file until it has found a value for each of its objects. Conceptually, the only information required for recovery is a copy of the committed version of each object in the server. This would be the most compact form for the recovery file. The name *checkpointing* is used to refer to the process of writing the current committed values of a server's objects to a new recovery file, together with transaction status entries and intentions lists of transactions that have not yet been fully resolved (including information related to the two-phase commit protocol). The term *checkpoint* is used to refer to the information stored by the checkpointing process. The purpose of making checkpoints is to reduce the number of transactions to be dealt with during recovery and to reclaim file space.

Checkpointing can be done immediately after recovery but before any new transactions are started. However, recovery may not occur very often. Therefore, checkpointing may need to be done from time to time during the normal activity of a server. The checkpoint is written to a future recovery file, and the current recovery file remains in use until the checkpoint is complete. Checkpointing consists of 'adding a mark' to the recovery file when the checkpointing starts, writing the server's objects to the future recovery file and then copying to that file (1) all entries before the mark that relate to as-yet-unresolved transactions and (2) all entries after the mark in the recovery file. When the checkpoint is complete, the future recovery file becomes the recovery file.

The recovery system can reduce its use of space by discarding the old recovery file. When the recovery manager is carrying out the recovery process, it may encounter a checkpoint in the recovery file. When this happens, it can immediately restore all outstanding objects from the checkpoint.

Figure 17.20 Shadow versions

	Map at start			*Map when T commits*			
	$A \rightarrow P_0$			$A \rightarrow P_1$			
	$B \rightarrow P_0'$			$B \rightarrow P_2$			
	$C \rightarrow P_0''$			$C \rightarrow P_0''$			
	P_0	P_0'	P_0''	P_1	P_2	P_3	P_4
Version store	100	200	300	80	220	278	242
			Checkpoint				

17.6.2 Shadow versions

The logging technique records transaction status entries, intentions lists and objects all in the same file – the log. The *shadow versions* technique is an alternative way to organize a recovery file. It uses a *map* to locate versions of the server's objects in a file called a *version store*. The map associates the identifiers of the server's objects with the positions of their current versions in the version store. The versions written by each transaction are 'shadows' of the previous committed versions. As we shall see, the transaction status entries and intentions lists are stored separately. Shadow versions are described first.

When a transaction is prepared to commit, any of the objects changed by the transaction are appended to the version store, leaving the corresponding committed versions unchanged. These new as-yet-tentative versions are called *shadow* versions. When a transaction commits, a new map is made by copying the old map and entering the positions of the shadow versions. To complete the commit process, the new map replaces the old map.

To restore the objects when a server is replaced after a crash, its recovery manager reads the map and uses the information in the map to locate the objects in the version store.

Figure 17.20 illustrates this technique with the same example involving transactions T and U . The first column in the table shows the map before transactions T and U, when the balances of the accounts A, B and C are \$100, \$200 and \$300, respectively. The second column shows the map after transaction T has committed.

The version store contains a checkpoint, followed by the versions of A and B at P_1 and P_2 made by transaction T. It also contains the shadow versions of B and C made by transaction U, at P_3 and P_4.

The map must always be written to a well-known place (for example, at the start of the version store or a separate file) so that it can be found when the system needs to be recovered.

The switch from the old map to the new map must be performed in a single atomic step. To achieve this it is essential that stable storage is used for the map, so that there is guaranteed to be a valid map even when a file write operation fails. The shadow versions method provides faster recovery than logging because the positions of the current committed objects are recorded in the map, whereas recovery from a log requires

searching throughout the log for objects. Logging should be faster than shadow versions during the normal activity of the system, though. This is because logging requires only a sequence of append operations to the same file, whereas shadow versions require an additional stable storage write (involving two unrelated disk blocks).

Shadow versions on their own are not sufficient for a server that handles distributed transactions. Transaction status entries and intentions lists are saved in a file called the *transaction status file*. Each intentions list represents the part of the map that will be altered by a transaction when it commits. The transaction status file may, for example, be organized as a log.

The figure below shows the map and the transaction status file for our current example when *T* has committed and *U* is prepared to commit:

Map
$A \rightarrow P_1$
$B \rightarrow P_2$
$C \rightarrow P_0$"

Transaction status file (stable storage)

T	*T*	*U*
prepared	committed	prepared
$A \rightarrow P_1$		$B \rightarrow P_3$
$B \rightarrow P_2$		$C \rightarrow P_4$

There is a chance that a server may crash between the time when a *committed* status is written to the transaction status file and the time when the map is updated – in which case the client will not have been acknowledged. The recovery manager must allow for this possibility when the server is replaced after a crash, for example by checking whether the map includes the effects of the last committed transaction in the transaction status file. If it does not, then the latter should be marked as aborted.

17.6.3 The need for transaction status and intentions list entries in a recovery file

It is possible to design a simple recovery file that does not include entries for transaction status items and intentions lists. This sort of recovery file may be suitable when all transactions are directed to a single server. The use of transaction status items and intentions lists in the recovery file is essential for a server that is intended to participate in distributed transactions. This approach can also be useful for servers of non-distributed transactions for various reasons, including the following:

- Some recovery managers are designed to write the objects to the recovery file early, under the assumption that transactions normally commit.
- If transactions use a large number of big objects, the need to write them contiguously to the recovery file may complicate the design of a server. When objects are referenced from intentions lists, they can be found wherever they are.
- In timestamp ordering concurrency control, a server sometimes knows that a transaction will eventually be able to commit and acknowledges the client – at this time, the objects are written to the recovery file (see Chapter 16) to ensure their permanence. However, the transaction may have to wait to commit until earlier transactions have committed. In such situations, the corresponding transaction status entries in the recovery file will be *waiting to commit* and then *committed* to

ensure timestamp ordering of committed transactions in the recovery file. On recovery, any waiting-to-commit transactions can be allowed to commit, because the ones they were waiting for will have either just committed or been aborted due to failure of the server.

17.6.4 Recovery of the two-phase commit protocol

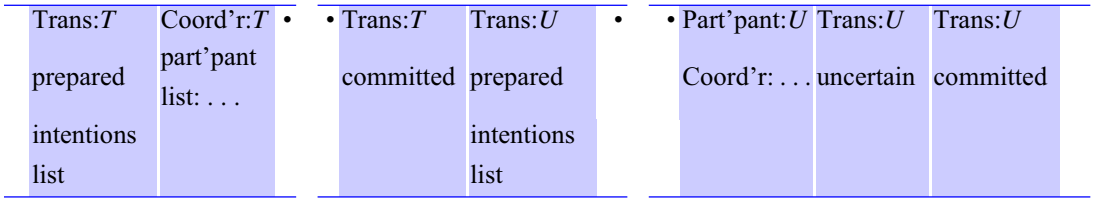
In a distributed transaction, each server keeps its own recovery file. The recovery management described in the previous section must be extended to deal with any transactions that are performing the two-phase commit protocol at the time when a server fails. The recovery managers use two new status values for this purpose: *done* and *uncertain*. These status values are shown in Figure 17.6. A coordinator uses *committed* to indicate that the outcome of the vote is *Yes* and *done* to indicate that the two-phase commit protocol is complete. A participant uses *uncertain* to indicate that it has voted *Yes* but does not yet know the outcome of the vote. Two additional types of entry allow a coordinator to record a list of participants and a participant to record its coordinator:

Type of entry	*Description of contents of entry*
Coordinator	Transaction identifier, list of participants
Participant	Transaction identifier, coordinator

In phase 1 of the protocol, when the coordinator is prepared to commit (and has already added a *prepared* status entry to its recovery file), its recovery manager adds a *coordinator* entry to its recovery file. Before a participant can vote *Yes*, it must have already prepared to commit (and must have already added a *prepared* status entry to its recovery file). When it votes *Yes*, its recovery manager records a *participant* entry and adds an *uncertain* transaction status to its recovery file as a forced write. When a participant votes *No*, it adds an *abort* transaction status to its recovery file.

In phase 2 of the protocol, the recovery manager of the coordinator adds either a *committed* or an *aborted* transaction status to its recovery file, according to the decision. This must be a forced write (that is, it is written immediately to the recovery file). Recovery managers of participants add a *commit* or *abort* transaction status to their recovery files according to the message received from the coordinator. When a coordinator has received a confirmation from all of its participants, its recovery manager adds a *done* transaction status to its recovery file – this need not be forced. The *done* status entry is not part of the protocol but is used when the recovery file is reorganized. Figure 17.21 shows the entries in a log for transaction *T*, in which the server played the coordinator role, and for transaction *U*, in which the server played the participant role. For both transactions, the *prepared* transaction status entry comes first. In the case of a coordinator it is followed by a coordinator entry and a *committed* transaction status entry. The *done* transaction status entry is not shown in Figure 17.21. In the case of a participant, the *prepared* transaction status entry is followed by a participant entry whose state is *uncertain* and then a *committed* or *aborted* transaction status entry.

Figure 17.21 Log with entries relating to two-phase commit protocol



When a server is replaced after a crash, the recovery manager has to deal with the two-phase commit protocol in addition to restoring the objects. For any transaction where the server has played the coordinator role, it should find a coordinator entry and a set of transaction status entries. For any transaction where the server played the participant role, it should find a participant entry and a set of transaction status entries. In both cases, the most recent transaction status entry – that is, the one nearest the end of the log – determines the transaction status at the time of failure. The action of the recovery manager with respect to the two-phase commit protocol for any transaction depends on whether the server was the coordinator or a participant and on its status at the time of failure, as shown in Figure 17.22.

Reorganization of recovery file • Care must be taken when performing a checkpoint to ensure that coordinator entries of transactions without status *done* are not removed from the recovery file. These entries must be retained until all the participants have confirmed that they have completed their transactions. Entries with status *done* may be discarded. Participant entries with transaction state *uncertain* must also be retained.

Recovery of nested transactions • In the simplest case, each subtransaction of a nested transaction accesses a different set of objects. As each participant prepares to commit during the two-phase commit protocol, it writes its objects and intentions lists to the local recovery file, associating them with the transaction identifier of the top-level transaction. Although nested transactions use a special variant of the two-phase commit protocol, the recovery manager uses the same transaction status values as for flat transactions.

However, abort recovery is complicated by the fact that several subtransactions at the same and different levels in the nesting hierarchy can access the same object. Section 16.4 describes a locking scheme in which parent transactions inherit locks and subtransactions acquire locks from their parents. The locking scheme forces parent transactions and subtransactions to access common data objects at different times and ensures that accesses by concurrent subtransactions to the same objects must be serialized.

Objects that are accessed according to the rules of nested transactions are made recoverable by providing tentative versions for each subtransaction. The relationship between the tentative versions of an object used by the subtransactions of a nested transaction is similar to the relationship between the locks. To support recovery from aborts, the server of an object shared by transactions at multiple levels provides a stack of tentative versions – one for each nested transaction to use.

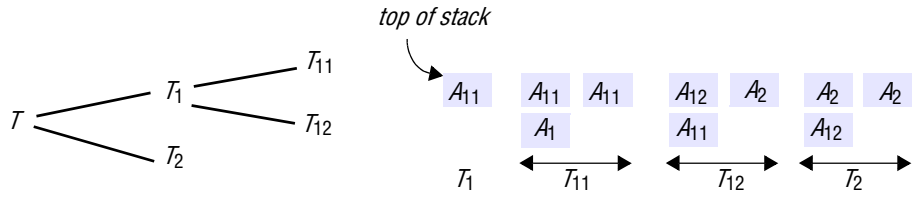
Figure 17.22 Recovery of the two-phase commit protocol

Role	*Status*	*Action of recovery manager*
Coordinator	*prepared*	No decision had been reached before the server failed. It sends *abortTransaction* to all the servers in the participant list and adds the transaction status *aborted* in its recovery file. Same action for state *aborted*. If there is no participant list, the participants will eventually time out and abort the transaction.
Coordinator	*committed*	A decision to commit had been reached before the server failed. It sends a *doCommit* to all of the participants in its participant list (in case it had not done so before) and resumes the two-phase protocol at step 4 (see Figure 17.5).
Participant	*committed*	The participant sends a *haveCommitted* message to the coordinator (in case this was not done before it failed). This will allow the coordinator to discard information about this transaction at the next checkpoint.
Participant	*uncertain*	The participant failed before it knew the outcome of the transaction. It cannot determine the status of the transaction until the coordinator informs it of the decision. It sends a *getDecision* to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.
Participant	*prepared*	The participant has not yet voted and can abort the transaction.
Coordinator	*done*	No action is required.

When the first subtransaction in a set of nested transactions accesses an object, it is provided with a tentative version that is a copy of the current committed version of the object. This is regarded as being at the top of the stack, but unless any of its subtransactions access the same object, the stack will not materialize.

When one of its subtransactions does access the same object, it copies the version at the top of the stack and pushes it back onto the stack. All of that subtransaction's updates are applied to the tentative version at the top of the stack. When a subtransaction provisionally commits, its parent inherits the new version. To achieve this, both the subtransaction's version and its parent's version are discarded from the stack and then the subtransaction's new version is pushed back onto the stack (effectively replacing its parent's version). When a subtransaction aborts, its version at the top of the stack is discarded. Eventually, when the top-level transaction commits, the version at the top of the stack (if any) becomes the new committed version.

Figure 17.23 Nested transactions



For example, in Figure 17.23, suppose that transactions T_1, T_{11}, T_{12} and T_2 all access the same object, A, in the order T_1; T_{11}; T_{12}; T_2. Suppose that their tentative versions are called A_1, A_{11}, A_{12} and A_2. When T_1 starts executing, A_1 is based on the committed version of A and is pushed onto the stack. When T_{11} starts executing, it bases its version A_{11} on A_1 and pushes it onto the stack; when it completes, it replaces its parent's version on the stack. Transactions T_{12} and T_2 act in a similar way, finally leaving the result of T_2 at the top of the stack.

18.1 Introduction

In this chapter, we study the replication of data: the maintenance of copies of data at multiple computers. Replication is a key to the effectiveness of distributed systems in that it can provide enhanced performance, high availability and fault tolerance. Replication is used widely. For example, the caching of resources from web servers in browsers and web proxy servers is a form of replication, since the data held in caches and at servers are replicas of one another. The DNS naming service, described in Chapter 13, maintains copies of name-to-attribute mappings for computers and is relied on for day-to-day access to services across the Internet.

Replication is a technique for enhancing services. The motivations for replication include:

Performance enhancement: The caching of data at clients and servers is by now familiar as a means of performance enhancement. For example, Chapter 2 pointed out that browsers and proxy servers cache copies of web resources to avoid the latency of fetching resources from the originating server. Furthermore, data are sometimes replicated transparently between several originating servers in the same domain. The workload is shared between the servers by binding all the server IP addresses to the site's DNS name, say *www.aWebSite.org*. A DNS lookup of *www.aWebSite.org* results in one of the several servers' IP addresses being returned, in a round-robin fashion (see Section 13.2.3). More sophisticated load-balancing strategies are required for more complex services based on data replicated between thousands of servers. As an example, Dilley *et al*. [2002] describe the approach to DNS name resolution adopted in the Akamai content distribution network.

Replication of immutable data is trivial: it increases performance with little cost to the system. Replication of changing data, such as that of the Web, incurs overheads in the form of protocols designed to ensure that clients receive up-to-date data (see Section 2.3.1). Thus there are limits to the effectiveness of replication as a performance-enhancement technique.

Increased availability: Users require services to be highly available. That is, the proportion of time for which a service is accessible with reasonable response times should be close to 100%. Apart from delays due to pessimistic concurrency control conflicts (due to data locking), the factors that are relevant to high availability are:

- server failures;
- network partitions and disconnected operation (communication disconnections that are often unplanned and are a side effect of user mobility).

To take the first of these, replication is a technique for automatically maintaining the availability of data despite server failures. If data are replicated at two or more failure-independent servers, then client software may be able to access data at an alternative server should the default server fail or become unreachable. That is, the percentage of time during which the *service* is available can be enhanced by replicating server data. If each of n servers has an independent probability p of failing

or becoming unreachable, then the availability of an object stored at each of these servers is:

$$1 - probability(all\ managers\ failed\ or\ unreachable) = 1 - p^n$$

For example, if there is a 5% probability of any individual server failing over a given time period and if there are two servers, then the availability is $1 - 0.05^2 = 1 - 0.0025 = 99.75\%$. An important difference between caching systems and server replication is that caches do not necessarily hold collections of objects such as files in their entirety. So caching does not necessarily enhance availability at the application level – a user may have one needed file but not another.

Network partitions (see Section 15.1) and disconnected operation are the second factor that militate against high availability. Mobile users may deliberately disconnect their computers or become unintentionally disconnected from a wireless network as they move around. For example, a user on a train with a laptop may have no access to networking (wireless networking may be interrupted, or they may have no such capability). In order to be able to work in these circumstances – so-called *disconnected working* or *disconnected operation* – the user will often prepare by copying heavily used data, such as the contents of a shared diary, from their usual environment to the laptop. But there is often a trade-off to availability during such a period of disconnection: when the user consults or updates the diary, they risk reading data that someone else has altered in the meantime. For example, they may make an appointment in a slot that has since been occupied. Disconnected working is only feasible if the user (or the application, on the user's behalf) can cope with stale data and can later resolve any conflicts that arise.

Fault tolerance: Highly available data is not necessarily strictly correct data. It may be out of date, for example; or two users on opposite sides of a network partition may make updates that conflict and need to be resolved. A fault-tolerant service, by contrast, always guarantees strictly correct behaviour despite a certain number and type of faults. The correctness concerns the freshness of data supplied to the client and the effects of the client's operations upon the data. Correctness sometimes also concerns the timeliness of the service's responses – such as, for example, in the case of a system for air traffic control, where correct data are needed on short timescales.

The same basic technique used for high availability – that of replicating data and functionality between computers – is also applicable for achieving fault tolerance. If up to f of $f + 1$ servers crash, then in principle at least one remains to supply the service. And if up to f servers can exhibit Byzantine failures, then in principle a group of $2f + 1$ servers can provide a correct service, by having the correct servers outvote the failed servers (who may supply spurious values). But fault tolerance is subtler than this simple description makes it seem. The system must manage the coordination of its components precisely to maintain the correctness guarantees in the face of failures, which may occur at any time.

A common requirement when data are replicated is for *replication transparency*. That is, clients should not normally have to be aware that multiple *physical* copies of data exist. As far as clients are concerned, data are organized as individual *logical* objects and they identify only one item in each case when they request an operation to be performed. Furthermore, clients expect operations to return only one set of values. This

is despite the fact that operations may be performed upon more than one physical copy in concert.

The other general requirement for replicated data – one that can vary in strength between applications – is that of consistency. This concerns whether the operations performed upon a collection of replicated objects produce results that meet the specification of correctness for those objects.

We saw in the example of the diary that during disconnected operation data may be allowed to become inconsistent, at least temporarily. But when clients remain connected it is often not acceptable for different clients (using different physical copies of data) to obtain inconsistent results when they make requests affecting the same logical objects. That is, it is not acceptable if the results break the application's correctness criteria.

We now examine in more detail the design issues raised when we replicate data to achieve highly available and fault-tolerant services. We also examine some standard solutions and techniques for dealing with those issues. First, Sections 18.2 to 18.4 cover the case where clients make individual invocations upon shared data. Section 18.2 presents a general architecture for managing replicated data and introduces group communication as an important tool. Group communication is particularly useful for achieving fault tolerance, which is the subject of Section 18.3. Section 18.4 describes techniques for high availability, including disconnected operation. It includes case studies of the gossip architecture, Bayou and the Coda file system. Section 18.5 examines how to support transactions on replicated data. As Chapters 16 and 17 explained, transactions are made up of sequences of operations, rather than single operations.