

## TRANSACTIONS AND CONCURRENCY CONTROL

- 16.1 Introduction
- 16.2 Transactions
- 16.3 Nested transactions
- 16.4 Locks
- 16.5 Optimistic concurrency control
- 16.6 Timestamp ordering
- 16.7 Comparison of methods for concurrency control
- 16.8 Summary

This chapter discusses the application of transactions and concurrency control to shared objects managed by servers.

A transaction defines a sequence of server operations that is guaranteed by the server to be atomic in the presence of multiple clients and server crashes. Nested transactions are structured from sets of other transactions. They are particularly useful in distributed systems because they allow additional concurrency.

All of the concurrency control protocols are based on the criterion of serial equivalence and are derived from rules for conflicts between operations. Three methods are described:

- Locks are used to order transactions that access the same objects according to the order of arrival of their operations at the objects.
- Optimistic concurrency control allows transactions to proceed until they are ready to commit, whereupon a check is made to see whether they have performed conflicting operations on objects.
- Timestamp ordering uses timestamps to order transactions that access the same objects according to their starting times.

## 16.1 Introduction

The goal of transactions is to ensure that all of the objects managed by a server remain in a consistent state when they are accessed by multiple transactions and in the presence of server crashes. Chapter 2 introduced a failure model for distributed systems. Transactions deal with crash failures of processes and omission failures in communication, but not any type of arbitrary (or Byzantine) behaviour. The failure model for transactions is presented in Section 16.1.2.

Objects that can be recovered after their server crashes are called *recoverable* objects. In general, the objects managed by a server may be stored in volatile memory (for example, RAM) or persistent memory (for example, a hard disk). Even if objects are stored in volatile memory, the server may use persistent memory to store sufficient information for the state of the objects to be recovered if the server process crashes. This enables servers to make objects recoverable. A transaction is specified by a client as a set of operations on objects to be performed as an indivisible unit by the servers managing those objects. The servers must guarantee that either the entire transaction is carried out and the results recorded in permanent storage or, in the case that one or more of them crashes, its effects are completely erased. The next chapter discusses issues related to transactions that involve several servers, in particular how they decide on the outcome of a distributed transaction. This chapter concentrates on the issues for a transaction at a single server. A client's transaction is also regarded as indivisible from the point of view of other clients' transactions in the sense that the operations of one transaction cannot observe the partial effects of the operations of another. Section 16.1.1 discusses simple synchronization of access to objects, and Section 16.2 introduces transactions, which require more advanced techniques to prevent interference between clients. Section 16.3 discusses nested transactions. Sections 16.4 to 16.6 discuss three methods of concurrency control for transactions whose operations are all addressed to a single server (locks, optimistic concurrency control and timestamp ordering). Chapter 17 discusses how these methods are extended for use with transactions whose operations are addressed to several servers.

To explain some of the points made in this chapter, we use a banking example, shown in Figure 16.1. Each account is represented by a remote object whose interface, *Account*, provides operations for making deposits and withdrawals and for enquiring about and setting the balance. Each branch of the bank is represented by a remote object whose interface, *Branch*, provides operations for creating a new account, for looking up an account by name and for enquiring about the total funds at that branch.

### 16.1.1 Simple synchronization (without transactions)

One of the main issues of this chapter is that unless a server is carefully designed, its operations performed on behalf of different clients may sometimes interfere with one another. Such interference may result in incorrect values in the objects. In this section, we discuss how client operations may be synchronized without recourse to transactions.

**Atomic operations at the server** • We have seen in earlier chapters that the use of multiple threads is beneficial to performance in many servers. We have also noted that the use of threads allows operations from multiple clients to run concurrently and

Figure 16.1 Operations of the *Account* interface

---

```

deposit(amount)
    deposit amount in the account

withdraw(amount)
    withdraw amount from the account

getBalance() → amount
    return the balance of the account

setBalance(amount)
    set the balance of the account to amount

```

---

Operations of the *Branch* interface

```

create(name) → account
    create a new account with a given name

lookUp(name) → account
    return a reference to the account with the given name

branchTotal() → amount
    return the total of all the balances at the branch

```

---

possibly access the same objects. Therefore, the methods of objects should be designed for use in a multi-threaded context. For example, if the methods *deposit* and *withdraw* are not designed for use in a multi-threaded program, then it is possible that the actions of two or more concurrent executions of the method could be interleaved arbitrarily and have strange effects on the instance variables of the account objects.

Chapter 7 explains the use of the *synchronized* keyword, which can be applied to methods in Java to ensure that only one thread at a time can access an object. In our example, the class that implements the *Account* interface will be able to declare the methods as synchronized. For example:

```

    public synchronized void deposit(int amount) throws RemoteException{
        // adds amount to the balance of the account
    }

```

If one thread invokes a synchronized method on an object, then that object is effectively locked, and another thread that invokes one of its synchronized methods will be blocked until the lock is released. This form of synchronization forces the execution of threads to be separated in time and ensures that the instance variables of a single object are accessed in a consistent manner. Without synchronization, two separate *deposit* invocations might read the balance before either has incremented it – resulting in an incorrect value. Any method that accesses an instance variable that can vary should be synchronized.

Operations that are free from interference from concurrent operations being performed in other threads are called *atomic operations*. The use of synchronized

methods in Java is one way of achieving atomic operations. But in other programming environments for multi-threaded servers the operations on objects still need to have atomic operations in order to keep their objects consistent. This may be achieved by the use of any available mutual exclusion mechanism, such as a mutex.

**Enhancing client cooperation by synchronization of server operations** • Clients may use a server as a means of sharing some resources. This is achieved by some clients using operations to update the server's objects and other clients using operations to access them. The above scheme for synchronized access to objects provides all that is required in many applications – it prevents threads interfering with one another. However, some applications require a way for threads to communicate with each other.

For example, a situation may arise in which the operation requested by one client cannot be completed until an operation requested by another client has been performed. This can happen when some clients are producers and others are consumers – the consumers may have to wait until a producer has supplied some more of the commodity in question. It can also occur when clients are sharing a resource – clients needing the resource may have to wait for other clients to release it. We shall see later in this chapter that a similar situation arises when locks or timestamps are used for concurrency control in transactions.

The Java *wait* and *notify* methods introduced in Chapter 7 allow threads to communicate with one another in a manner that solves the above problems. They must be used within synchronized methods of an object. A thread calls *wait* on an object so as to suspend itself and to allow another thread to execute a method of that object. A thread calls *notify* to inform any thread waiting on that object that it has changed some of its data. Access to an object is still atomic when threads wait for one another: a thread that calls *wait* gives up its lock and suspends itself as a single atomic action; when a thread is restarted after being notified it acquires a new lock on the object and resumes execution from after its *wait*. A thread that calls *notify* (from within a synchronized method) completes the execution of that method before releasing the lock on the object.

Consider the implementation of a shared *Queue* object with two methods: *first* removes and returns the first object in the queue, and *append* adds a given object to the end of the queue. The method *first* will test whether the queue is empty, in which case it will call *wait* on the queue. If a client invokes *first* when the queue is empty, it will not get a reply until another client has added something to the queue – the *append* operation will call *notify* when it has added an object to the queue. This allows one of the threads waiting on the queue object to resume and to return the first object in the queue to its client. When threads can synchronize their actions on an object by means of *wait* and *notify*, the server holds onto requests that cannot immediately be satisfied and the client waits for a reply until another client has produced whatever it needs.

In Section 16.4, we discuss the implementation of a lock as an object with synchronized operations. When clients attempt to acquire a lock, they can be made to wait until the lock is released by other clients.

Without the ability to synchronize threads in this way, a client that cannot be satisfied immediately – for example, a client that invokes the *first* method on an empty queue – is told to try again later. This is unsatisfactory, because it will involve the client in polling the server and the server in carrying out extra requests. It is also potentially unfair because other clients may make their requests before the waiting client tries again.

### 16.1.2 Failure model for transactions

Lampson [1981] proposed a fault model for distributed transactions that accounts for failures of disks, servers and communication. In this model, the claim is that the algorithms work correctly in the presence of predictable faults, but no claims are made about their behaviour when a disaster occurs. Although errors may occur, they can be detected and dealt with before any incorrect behaviour results. The model states the following:

- Writes to permanent storage may fail, either by writing nothing or by writing a wrong value – for example, writing to the wrong block is a disaster. File storage may also decay. Reads from permanent storage can detect (by a checksum) when a block of data is bad.
- Servers may crash occasionally. When a crashed server is replaced by a new process, its volatile memory is first set to a state in which it knows none of the values (for example, of objects) from before the crash. After that it carries out a recovery procedure using information in permanent storage and obtained from other processes to set the values of objects including those related to the two-phase commit protocol (see Section 17.6). When a processor is faulty, it is made to crash so that it is prevented from sending erroneous messages and from writing wrong values to permanent storage – that is, so it cannot produce arbitrary failures. Crashes can occur at any time; in particular, they may occur during recovery.
- There may be an arbitrary delay before a message arrives. A message may be lost, duplicated or corrupted. The recipient can detect corrupted messages using a checksum. Both forged messages and undetected corrupt messages are regarded as disasters.

The fault model for permanent storage, processors and communications was used to design a stable system whose components can survive any single fault and present a simple failure model. In particular, *stable storage* provided an atomic *write* operation in the presence of a single fault of the *write* operation or a crash failure of the process. This was achieved by replicating each block on two disk blocks. A *write* operation was applied to the pair of disk blocks, and in the case of a single fault, one good block was always available. A *stable processor* used stable storage to enable it to recover its objects after a crash. Communication errors were masked by using a reliable remote procedure calling mechanism.

## 16.2 Transactions

In some situations, clients require a sequence of separate requests to a server to be atomic in the sense that:

1. They are free from interference by operations being performed on behalf of other concurrent clients.
2. Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

Figure 16.2 A client's banking transaction

```
Transaction T:  
a.withdraw(100);  
b.deposit(100);  
c.withdraw(200);  
b.deposit(200);
```

---

We return to our banking example to illustrate transactions. A client that performs a sequence of operations on a particular bank account on behalf of a user will first *lookUp* the account by name and then apply the *deposit*, *withdraw* and *getBalance* operations directly to the relevant account. In our examples, we use accounts with names *A*, *B* and *C*. The client looks them up and stores references to them in variables *a*, *b* and *c* of type *Account*. The details of looking up the accounts by name and the declarations of the variables are omitted from the examples.

Figure 16.2 shows an example of a simple client transaction specifying a series of related actions involving the bank accounts *A*, *B* and *C*. The first two actions transfer \$100 from *A* to *B* and the second two transfer \$200 from *C* to *B*. A client achieves a transfer operation by doing a withdrawal followed by a deposit.

Transactions originate from database management systems. In that context, a transaction is an execution of a program that accesses a database. Transactions were introduced to distributed systems in the form of transactional file servers such as XDIFS [Mitchell and Dion 1982]. In the context of a transactional file server, a transaction is an execution of a sequence of client requests for file operations. Transactions on distributed objects were provided in several research systems, including Argus [Liskov 1988] and Arjuna [Shrivastava *et al.* 1991]. In this last context, a transaction consists of the execution of a sequence of client requests such as, for example, those in Figure 16.2. From the client's point of view, a transaction is a sequence of operations that forms a single step, transforming the server data from one consistent state to another.

Transactions can be provided as a part of middleware. For example, CORBA provides the specification for an Object Transaction Service [OMG 2003] with IDL interfaces allowing clients' transactions to include multiple objects at multiple servers. The client is provided with operations to specify the beginning and end of a transaction. The client maintains a context for each transaction, which it propagates with each operation in that transaction. In CORBA, transactional objects are invoked within the scope of a transaction and generally have some persistent store associated with them.

In all of these contexts, a transaction applies to recoverable objects and is intended to be atomic. It is often called an *atomic transaction*. There are two aspects to atomicity:

**All or nothing:** A transaction either completes successfully, in which case the effects of all of its operations are recorded in the objects, or (if it fails or is deliberately aborted) has no effect at all. This all-or-nothing effect has two further aspects of its own:

*Failure atomicity:* The effects are atomic even when the server crashes.

**Durability:** After a transaction has completed successfully, all its effects are saved in permanent storage. We use the term ‘permanent storage’ to refer to files held on disk or another permanent medium. Data saved in a file will survive if the server process crashes.

**Isolation:** Each transaction must be performed without interference from other transactions; in other words, the intermediate effects of a transaction must not be visible to other transactions. The box below introduces a mnemonic, ACID, for remembering the properties of atomic transactions.

To support the requirement for failure atomicity and durability, the objects must be *recoverable*; that is, when a server process crashes unexpectedly due to a hardware fault or a software error, the changes due to all completed transactions must be available in permanent storage so that when the server is replaced by a new process, it can recover the objects to reflect the all-or-nothing effect. By the time a server acknowledges the completion of a client’s transaction, all of the transaction’s changes to the objects must have been recorded in permanent storage.

A server that supports transactions must synchronize the operations sufficiently to ensure that the isolation requirement is met. One way of doing this is to perform the transactions serially – one at a time, in some arbitrary order. Unfortunately, this solution would generally be unacceptable for servers whose resources are shared by multiple interactive users. For instance, in our banking example it is desirable to allow several bank clerks to perform online banking transactions at the same time as one another.

The aim for any server that supports transactions is to maximize concurrency. Therefore transactions are allowed to execute concurrently if this would have the same effect as a serial execution – that is, if they are *serially equivalent* or *serializable*.

### ACID properties

Härder and Reuter [1983] suggested the mnemonic ‘ACID’ to remember the properties of transactions, which are as follows:

**Atomicity:** a transaction must be all or nothing;

**Consistency:** a transaction takes the system from one consistent state to another consistent state;

**Isolation;**

**Durability.**

We have not included ‘consistency’ in our list of the properties of transactions because it is generally the responsibility of the programmers of servers and clients to ensure that transactions leave the database consistent.

As an example of consistency, suppose that in the banking example, an object holds the sum of all the account balances and its value is used as the result of *branchTotal*. Clients can get the sum of all the account balances either by using *branchTotal* or by calling *getBalance* on each of the accounts. For consistency, they should get the same result from both methods. To maintain this consistency, the *deposit* and *withdraw* operations must update the object holding the sum of all the account balances.

Figure 16.3 Operations in the *Coordinator* interface

*openTransaction()* → *trans*;

Starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

*closeTransaction(trans)* → (*commit*, *abort*);

Ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

*abortTransaction(trans)*;

Aborts the transaction.

Transaction capabilities can be added to servers of recoverable objects. Each transaction is created and managed by a coordinator, which implements the *Coordinator* interface shown in Figure 16.3. The coordinator gives each transaction an identifier, or TID. The client invokes the *openTransaction* method of the coordinator to introduce a new transaction – a transaction identifier or TID is allocated and returned. At the end of a transaction, the client invokes the *closeTransaction* method to indicate its end – all of the recoverable objects accessed by the transaction should be saved. If, for some reason, the client wants to abort a transaction, it invokes the *abortTransaction* method – all of its effects should be removed from sight.

A transaction is achieved by cooperation between a client program, some recoverable objects and a coordinator. The client specifies the sequence of invocations on recoverable objects that are to comprise a transaction. To achieve this, the client sends with each invocation the transaction identifier returned by *openTransaction*. One way to make this possible is to include an extra argument in each operation of a recoverable object to carry the TID. For example, in the banking service the *deposit* operation might be defined:

*deposit(trans, amount)*

Deposits *amount* in the account for transaction with TID *trans*

When transactions are provided as middleware, the TID can be passed implicitly with all remote invocations between *openTransaction* and *closeTransaction* or *abortTransaction*. This is what the CORBA Transaction Service does. We shall not show TIDs in our examples.

Normally, a transaction completes when the client makes a *closeTransaction* request. If the transaction has progressed normally, the reply states that the transaction is *committed* – this constitutes a promise to the client that all of the changes requested in the transaction are permanently recorded and that any future transactions that access the same data will see the results of all of the changes made during the transaction.

Alternatively, the transaction may have to *abort* for one of several reasons related to the nature of the transaction itself, to conflicts with another transaction or to the crashing of a process or computer. When a transaction is aborted the parties involved (the recoverable objects and the coordinator) must ensure that none of its effects are visible to future transactions, either in the objects or in their copies in permanent storage.

A transaction is either successful or is aborted in one of two ways – the client aborts it (using an *abortTransaction* call to the server) or the server aborts it. Figure 16.4



Figure 16.4 Transaction life histories

| Successful              | Aborted by client       | Aborted by server                                   |
|-------------------------|-------------------------|---|
| <i>openTransaction</i>  | <i>openTransaction</i>  | <i>openTransaction</i>                              |
| <i>operation</i>        | <i>operation</i>        | <i>operation</i>                                    |
| <i>operation</i>        | <i>operation</i>        | <i>operation</i>                                    |
| •                       | •                       | server aborts •                                     |
| •                       | •                       | <i>transaction</i> → •                              |
| <i>operation</i>        | <i>operation</i>        | <i>operation ERROR</i><br><i>reported to client</i> |
| <i>closeTransaction</i> | <i>abortTransaction</i> |   |

shows these three alternative life histories for transactions. We refer to a transaction as *failing* in both of the latter cases.

**Service actions related to process crashes** • If a server process crashes unexpectedly, it is eventually replaced. The new server process aborts any uncommitted transactions and uses a recovery procedure to restore the values of the objects to the values produced by the most recently committed transaction. To deal with a client that crashes unexpectedly during a transaction, servers can give each transaction an expiry time and abort any transaction that has not completed before its expiry time.

**Client actions related to server process crashes** • If a server crashes while a transaction is in progress, the client will become aware of this when one of the operations returns an exception after a timeout. If a server crashes and is then replaced during the progress of a transaction, the transaction will no longer be valid and the client must be informed via an exception to the next operation. In either case, the client must then formulate a plan, possibly in consultation with the human user, for the completion or abandonment of the task of which the transaction was a part.

16.2.1 Concurrency control

This section illustrates two well-known problems of concurrent transactions in the context of the banking example – the ‘lost update’ problem and the ‘inconsistent retrievals’ problem. We then show how both of these problems can be avoided by using serially equivalent executions of transactions. We assume throughout that each of the operations *deposit*, *withdraw*, *getBalance* and *setBalance* is a synchronized operation – that is, that its effects on the instance variable that records the balance of an account are atomic.

**The lost update problem** • The lost update problem is illustrated by the following pair of transactions on bank accounts *A*, *B* and *C*, whose initial balances are \$100, \$200 and \$300, respectively. Transaction *T* transfers an amount from account *A* to account *B*. Transaction *U* transfers an amount from account *C* to account *B*. In both cases, the

Figure 16.5    The lost update problem

| Transaction T:   | Transaction U:   |
|--|--|
| <i>balance = b.getBalance();</i><br><i>b.setBalance(balance*1.1);</i><br><i>a.withdraw(balance/10)</i> | <i>balance = b.getBalance();</i><br><i>b.setBalance(balance*1.1);</i><br><i>c.withdraw(balance/10)</i> |
| <i>balance = b.getBalance();</i> \$200   | <i>balance = b.getBalance();</i> \$200   |
|  | <i>b.setBalance(balance*1.1);</i> \$220  |
| <i>b.setBalance(balance*1.1);</i> \$220  |  |
| <i>a.withdraw(balance/10)</i> \$80   | <i>c.withdraw(balance/10)</i> \$280  |

amount transferred is calculated to increase the balance of *B* by 10%. The net effects on account *B* of executing the transactions *T* and *U* should be to increase the balance of account *B* by 10% twice, so its final value is \$242.

Now consider the effects of allowing the transactions *T* and *U* to run concurrently, as in Figure 16.5. Both transactions get the balance of *B* as \$200 and then deposit \$20. The result is incorrect, increasing the balance of account *B* by \$20 instead of \$42. This is an illustration of the ‘lost update’ problem. *U*’s update is lost because *T* overwrites it without seeing it. Both transactions have read the old value before either writes the new value.

In Figure 16.5 onwards, we show the operations that affect the balance of an account on successive lines down the page, and the reader should assume that an operation on a particular line is executed at a later time than the one on the line above it.

**Inconsistent retrievals** • Figure 16.6 shows another example related to a bank account in which transaction *V* transfers a sum from account *A* to *B* and transaction *W* invokes the *branchTotal* method to obtain the sum of the balances of all the accounts in the bank.

Figure 16.6    The inconsistent retrievals problem

| Transaction V:                                  | Transaction W:                              |
|---|---|
| <i>a.withdraw(100)</i><br><i>b.deposit(100)</i> | <i>aBranch.branchTotal()</i>                |
| <i>a.withdraw(100);</i> \$100                   | <i>total = a.getBalance()</i> \$100         |
|   | <i>total = total + b.getBalance()</i> \$300 |
|   | <i>total = total + c.getBalance()</i>       |
| <i>b.deposit(100)</i> \$300                     | •<br>•                                      |

Figure 16.7 A serially equivalent interleaving of *T* and *U*

| Transaction <i>T</i> :                     |       | Transaction <i>U</i> :                     |       |
|--|-------|--|-------|
| <i>balance</i> = <i>b.getBalance</i> ()    |       | <i>balance</i> = <i>b.getBalance</i> ()    |       |
| <i>b.setBalance</i> ( <i>balance</i> *1.1) |       | <i>b.setBalance</i> ( <i>balance</i> *1.1) |       |
| <i>a.withdraw</i> ( <i>balance</i> /10)    |       | <i>c.withdraw</i> ( <i>balance</i> /10)    |       |
| <i>balance</i> = <i>b.getBalance</i> ()    | \$200 | <i>balance</i> = <i>b.getBalance</i> ()    | \$220 |
| <i>b.setBalance</i> ( <i>balance</i> *1.1) | \$220 | <i>b.setBalance</i> ( <i>balance</i> *1.1) | \$242 |
| <i>a.withdraw</i> ( <i>balance</i> /10)    | \$80  | <i>c.withdraw</i> ( <i>balance</i> /10)    | \$278 |

The balances of the two bank accounts, *A* and *B*, are both initially \$200. The result of *branchTotal* includes the sum of *A* and *B* as \$300, which is wrong. This is an illustration of the ‘inconsistent retrievals’ problem. *W*’s retrievals are inconsistent because *V* has performed only the withdrawal part of a transfer at the time the sum is calculated.

**Serial equivalence** • If each of several transactions is known to have the correct effect when it is done on its own, then we can infer that if these transactions are done one at a time in some order the combined effect will also be correct. An interleaving of the operations of transactions in which the combined effect is the same as if the transactions had been performed one at a time in some order is a *serially equivalent* interleaving. When we say that two different transactions have the *same effect* as one another, we mean that the *read* operations return the same values and that the instance variables of the objects have the same values at the end.

The use of serial equivalence as a criterion for correct concurrent execution prevents the occurrence of lost updates and inconsistent retrievals.

The lost update problem occurs when two transactions read the old value of a variable and then use it to calculate the new value. This cannot happen if one transaction is performed before the other, because the later transaction will read the value written by the earlier one. As a serially equivalent interleaving of two transactions produces the same effect as a serial one, we can solve the lost update problem by means of serial equivalence. Figure 16.7 shows one such interleaving in which the operations that affect the shared account, *B*, are actually serial, for transaction *T* does all its operations on *B* before transaction *U* does. Another interleaving of *T* and *U* that has this property is one in which transaction *U* completes its operations on account *B* before transaction *T* starts.

We now consider the effect of serial equivalence in relation to the inconsistent retrievals problem, in which transaction *V* is transferring a sum from account *A* to *B* and transaction *W* is obtaining the sum of all the balances (see Figure 16.6). The inconsistent retrievals problem can occur when a retrieval transaction runs concurrently with an update transaction. It cannot occur if the retrieval transaction is performed before or after the update transaction. A serially equivalent interleaving of a retrieval transaction and an update transaction, for example as in Figure 16.8, will prevent inconsistent retrievals occurring.

Figure 16.8     A serially equivalent interleaving of V and W

| Transaction V:                                   |       | Transaction W:                         |       |
|--|-------|--|-------|
| <i>a.withdraw(100);</i><br><i>b.deposit(100)</i> |       | <i>aBranch.branchTotal( )</i>          |       |
| <i>a.withdraw(100);</i>                          | \$100 | <i>total = a.getBalance( )</i>         | \$100 |
| <i>b.deposit(100)</i>                            | \$300 | <i>total = total + b.getBalance( )</i> | \$400 |
|  |       | <i>total = total + c.getBalance( )</i> |       |
|  |       | ...                                    |       |

**Conflicting operations** • When we say that a pair of operations *conflicts* we mean that their combined effect depends on the order in which they are executed. To simplify matters we consider a pair of operations, *read* and *write*. *read* accesses the value of an object and *write* changes its value. The *effect* of an operation refers to the value of an object set by a *write* operation and the result returned by a *read* operation. The conflict rules for *read* and *write* operations are given in Figure 16.9.

For any pair of transactions, it is possible to determine the order of pairs of conflicting operations on objects accessed by both of them. Serial equivalence can be defined in terms of operation conflicts as follows:

For two transactions to be *serially equivalent*, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access.

Figure 16.9     *Read* and *write* operation conflict rules

| Operations of different transactions |              | Conflict | Reason   |
|--------------------------------------|--------------|----------|--|
| <i>read</i>                          | <i>read</i>  | No       | Because the effect of a pair of <i>read</i> operations does not depend on the order in which they are executed |
| <i>read</i>                          | <i>write</i> | Yes      | Because the effect of a <i>read</i> and a <i>write</i> operation depends on the order of their execution       |
| <i>write</i>                         | <i>write</i> | Yes      | Because the effect of a pair of <i>write</i> operations depends on the order of their execution                |

Figure 16.10 A non-serially-equivalent interleaving of operations of transactions  $T$  and  $U$

| Transaction $T$ :     | Transaction $U$ :     |
|-----------------------|-----------------------|
| $x = \text{read}(i)$  |                       |
| $\text{write}(i, 10)$ |                       |
|                       | $y = \text{read}(j)$  |
|                       | $\text{write}(j, 30)$ |
| $\text{write}(j, 20)$ |                       |
|                       | $z = \text{read}(i)$  |

Consider as an example the transactions  $T$  and  $U$ , defined as follows:

```
T: x = read(i); write(i, 10); write(j, 20);
U: y = read(j); write(j, 30); z = read(i);
```

Then consider the interleaving of their executions, shown in Figure 16.10. Note that each transaction’s access to objects  $i$  and  $j$  is serialized with respect to one another, because  $T$  makes all of its accesses to  $i$  before  $U$  does and  $U$  makes all of its accesses to  $j$  before  $T$  does. But the ordering is not serially equivalent, because the pairs of conflicting operations are not done in the same order at both objects. Serially equivalent orderings require one of the following two conditions:

- 1.  $T$  accesses  $i$  before  $U$  and  $T$  accesses  $j$  before  $U$ .
- 2.  $U$  accesses  $i$  before  $T$  and  $U$  accesses  $j$  before  $T$ .

Serial equivalence is used as a criterion for the derivation of concurrency control protocols. These protocols attempt to serialize transactions in their access to objects. Three alternative approaches to concurrency control are commonly used: locking, optimistic concurrency control and timestamp ordering. However, most practical systems use locking, which is discussed in Section 16.4. When locking is used, the server sets a lock, labelled with the transaction identifier, on each object just before it is accessed and removes these locks when the transaction has completed. While an object is locked, only the transaction that it is locked for can access that object; other transactions must either wait until the object is unlocked or, in some cases, share the lock. The use of locks can lead to deadlocks, with transactions waiting for each other to release locks – for example, when a pair of transactions each has an object locked that the other needs to access. We discuss the deadlock problem and some remedies for it in Section 16.4.1.

Optimistic concurrency control is described in Section 16.5. In optimistic schemes, a transaction proceeds until it asks to commit, and before it is allowed to commit the server performs a check to discover whether it has performed operations on any objects that conflict with the operations of other concurrent transactions, in which case the server aborts it and the client may restart it. The aim of the check is to ensure that all the objects are correct.

Timestamp ordering is described in Section 16.6. In timestamp ordering, a server records the most recent time of reading and writing of each object and for each

Figure 16.11 A dirty read when transaction *T* aborts

| Transaction <i>T</i> :                  | Transaction <i>U</i> :                  |
|---|---|
| <i>a.getBalance()</i>                   | <i>a.getBalance()</i>                   |
| <i>a.setBalance(balance + 10)</i>       | <i>a.setBalance(balance + 20)</i>       |
| <i>balance = a.getBalance()</i> \$100   |   |
| <i>a.setBalance(balance + 10)</i> \$110 |   |
|   | <i>balance = a.getBalance()</i> \$110   |
|   | <i>a.setBalance(balance + 20)</i> \$130 |
|   | <i>commit transaction</i>               |
| <i>abort transaction</i>                |   |

operation, the timestamp of the transaction is compared with that of the object to determine whether it can be done immediately or must be delayed or rejected. When an operation is delayed, the transaction waits; when it is rejected, the transaction is aborted.

Basically, concurrency control can be achieved either by clients’ transactions waiting for one another or by restarting transactions after conflicts between operations have been detected, or by a combination of the two.

16.2.2 Recoverability from aborts

Servers must record all the effects of committed transactions and none of the effects of aborted transactions. They must therefore allow for the fact that a transaction may abort by preventing it affecting other concurrent transactions if it does so.

This section illustrates two problems associated with aborting transactions in the context of the banking example. These problems are called ‘dirty reads’ and ‘premature writes’, and both of them can occur in the presence of serially equivalent executions of transactions. These issues are concerned with the effects of operations on objects such as the balance of a bank account. To simplify things, operations are considered in two categories: *read* operations and *write* operations. In our illustrations, *getBalance* is a *read* operation and *setBalance* a *write* operation.

**Dirty reads** • The isolation property of transactions requires that transactions do not see the uncommitted state of other transactions. The ‘dirty read’ problem is caused by the interaction between a *read* operation in one transaction and an earlier *write* operation in another transaction on the same object. Consider the executions illustrated in Figure 16.11, in which *T* gets the balance of account *A* and sets it to \$10 more, then *U* gets the balance of account *A* and sets it to \$20 more, and the two executions are serially equivalent. Now suppose that the transaction *T* aborts after *U* has committed. Then the transaction *U* will have seen a value that never existed, since *A* will be restored to its original value. We say that the transaction *U* has performed a *dirty read*. As it has committed, it cannot be undone.

Figure 16.12 Overwriting uncommitted values

| Transaction <i>T</i> :   |       | Transaction <i>U</i> :   |       |
|--------------------------|-------|--------------------------|-------|
| <i>a.setBalance(105)</i> |       | <i>a.setBalance(110)</i> |       |
|                          | \$100 |                          |       |
| <i>a.setBalance(105)</i> | \$105 |                          |       |
|                          |       | <i>a.setBalance(110)</i> | \$110 |

**Recoverability of transactions** • If a transaction (like *U*) has committed after it has seen the effects of a transaction that subsequently aborted, the situation is not recoverable. To ensure that such situations will not arise, any transaction (like *U*) that is in danger of having a dirty read delays its commit operation. The strategy for recoverability is to delay commits until after the commitment of any other transaction whose uncommitted state has been observed. In our example, *U* delays its commit until after *T* commits. In the case that *T* aborts, then *U* must abort as well.

**Cascading aborts** • In Figure 16.11, suppose that transaction *U* delays committing until after *T* aborts. As we have said, *U* must abort as well. Unfortunately, if any other transactions have seen the effects due to *U*, they too must be aborted. The aborting of these latter transactions may cause still further transactions to be aborted. Such situations are called *cascading aborts*. To avoid cascading aborts, transactions are only allowed to read objects that were written by committed transactions. To ensure that this is the case, any *read* operation must be delayed until other transactions that applied a *write* operation to the same object have committed or aborted. The avoidance of cascading aborts is a stronger condition than recoverability.

**Premature writes** • Consider another implication of the possibility that a transaction may abort. This one is related to the interaction between *write* operations on the same object belonging to different transactions. For an illustration, we consider two *setBalance* transactions, *T* and *U*, on account *A*, as shown in Figure 16.12. Before the transactions, the balance of account *A* was \$100. The two executions are serially equivalent, with *T* setting the balance to \$105 and *U* setting it to \$110. If the transaction *U* aborts and *T* commits, the balance should be \$105.

Some database systems implement the action of *abort* by restoring ‘before images’ of all the *writes* of a transaction. In our example, *A* is \$100 initially, which is the ‘before image’ of *T*’s *write*; similarly, \$105 is the ‘before image’ of *U*’s *write*. Thus if *U* aborts, we get the correct balance of \$105.

Now consider the case when *U* commits and then *T* aborts. The balance should be \$110, but as the ‘before image’ of *T*’s *write* is \$100, we get the wrong balance of \$100. Similarly, if *T* aborts and then *U* aborts, the ‘before image’ of *U*’s *write* is \$105 and we get the wrong balance of \$105 – the balance should revert to \$100.

To ensure correct results in a recovery scheme that uses before images, *write* operations must be delayed until earlier transactions that updated the same objects have either committed or aborted.

**Strict executions of transactions** • Generally, it is required that transactions delay both their *read* and *write* operations so as to avoid both dirty reads and premature writes. The executions of transactions are called *strict* if the service delays both *read* and *write* operations on an object until all transactions that previously wrote that object have either committed or aborted. The strict execution of transactions enforces the desired property of isolation.

**Tentative versions** • For a server of recoverable objects to participate in transactions, it must be designed so that any updates of objects can be removed if and when a transaction aborts. To make this possible, all of the update operations performed during a transaction are done in tentative versions of objects in volatile memory. Each transaction is provided with its own private set of tentative versions of any objects that it has altered. All the update operations of a transaction store values in the transaction's own private set. Access operations in a transaction take values from the transaction's own private set if possible, or failing that, from the objects.

The tentative versions are transferred to the objects only when a transaction commits, by which time they will also have been recorded in permanent storage. This is performed in a single step, during which other transactions are excluded from access to the objects that are being altered. When a transaction aborts, its tentative versions are deleted.

## 16.3 Nested transactions

---

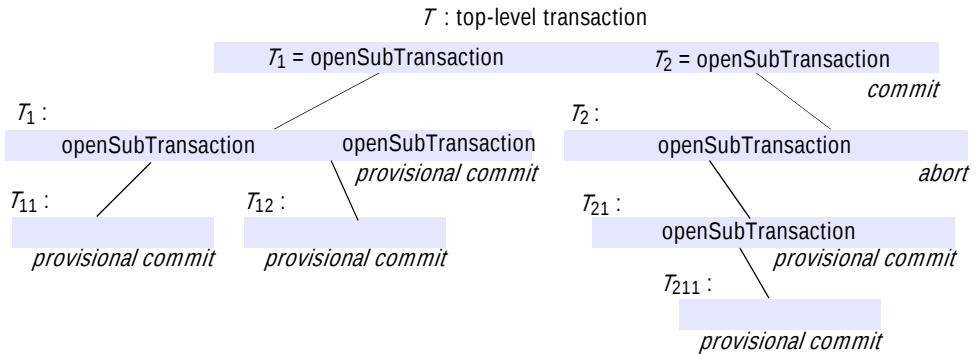
Nested transactions extend the above transaction model by allowing transactions to be composed of other transactions. Thus several transactions may be started from within a transaction, allowing transactions to be regarded as modules that can be composed as required.

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*. For example, in Figure 16.13,  $T$  is a top-level transaction that starts a pair of subtransactions,  $T_1$  and  $T_2$ . The subtransaction  $T_1$  starts its own pair of subtransactions,  $T_{11}$  and  $T_{22}$ . Also, subtransaction  $T_2$  starts its own subtransaction,  $T_{21}$ , which starts another subtransaction,  $T_{211}$ .

A subtransaction appears atomic to its parent with respect to transaction failures and to concurrent access. Subtransactions at the same level, such as  $T_1$  and  $T_2$ , can run concurrently, but their access to common objects is serialized – for example, by the locking scheme described in Section 16.4. Each subtransaction can fail independently of its parent and of the other subtransactions. When a subtransaction aborts, the parent transaction can sometimes choose an alternative subtransaction to complete its task. For example, a transaction to deliver a mail message to a list of recipients could be structured as a set of subtransactions, each of which delivers the message to one of the recipients. If one or more of the subtransactions fails, the parent transaction could record the fact and then commit, with the result that all the successful child transactions commit. It could then start another transaction to attempt to redeliver the messages that were not sent the first time.



Figure 16.13 Nested transactions



When we need to distinguish our original form of transaction from nested ones, we use the term *flat* transaction. It is flat because all of its work is done at the same level between an *openTransaction* and a *commit* or *abort*, and it is not possible to commit or abort parts of it. Nested transactions have the following main advantages:

1. Subtransactions at one level (and their descendants) may run concurrently with other subtransactions at the same level in the hierarchy. This can allow additional concurrency in a transaction. When subtransactions run in different servers, they can work in parallel. For example, consider the *branchTotal* operation in our banking example. It can be implemented by invoking *getBalance* at every account in the branch. Now each of these invocations may be performed as a subtransaction, in which case they can be performed concurrently. Since each one applies to a different account, there will be no conflicting operations among the subtransactions.
2. Subtransactions can commit or abort independently. In comparison with a single transaction, a set of nested subtransactions is potentially more robust. The above example of delivering mail shows that this is so – with a flat transaction, one transaction failure would cause the whole transaction to be restarted. In fact, a parent can decide on different actions according to whether a subtransaction has aborted or not.

The rules for committing of nested transactions are rather subtle:

- A transaction may commit or abort only after its child transactions have completed.
- When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. Its decision to abort is final.
- When a parent aborts, all of its subtransactions are aborted. For example, if  $T_2$  aborts then  $T_{21}$  and  $T_{211}$  must also abort, even though they may have provisionally committed.
- When a subtransaction aborts, the parent can decide whether to abort or not. In our example,  $T$  decides to commit although  $T_2$  has aborted.

- If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted. In our example,  $T$ 's commitment allows  $T_1$ ,  $T_{11}$  and  $T_{12}$  to commit, but not  $T_{21}$  and  $T_{211}$  since their parent,  $T_2$ , aborted. Note that the effects of a subtransaction are not permanent until the top-level transaction commits.

In some cases, the top-level transaction may decide to abort because one or more of its subtransactions have aborted. As an example, consider the following *Transfer* transaction:

```
Transfer $100 from  $B$  to  $A$ 
 $a.deposit(100)$ 
 $b.withdraw(100)$ 
```

This can be structured as a pair of subtransactions, one for the *withdraw* operation and the other for *deposit*. When the two subtransactions both commit, the *Transfer* transaction can also commit. Suppose that a *withdraw* subtransaction aborts whenever an account is overdrawn. Now consider the case when the *withdraw* subtransaction aborts and the *deposit* subtransaction commits – and recall that the commitment of a child transaction is conditional on the parent transaction committing. We presume that the top-level (*Transfer*) transaction will decide to abort. The aborting of the parent transaction causes the subtransactions to abort – so the *deposit* transaction is aborted and all its effects are undone.

The CORBA Object Transaction Service supports both flat and nested transactions. Nested transactions are particularly useful in distributed systems because child transactions may be run concurrently in different servers. We return to this issue in Chapter 17. This form of nested transactions is due to Moss [1985]. Other variants of nested transactions with different serializability properties have been proposed; for example, see Weikum [1991].

## 16.4 Locks

Transactions must be scheduled so that their effect on shared data is serially equivalent. A server can achieve serial equivalence of transactions by serializing access to the objects. Figure 16.7 shows an example of how serial equivalence can be achieved with some degree of concurrency – transactions  $T$  and  $U$  both access account  $B$ , but  $T$  completes its access before  $U$  starts accessing it.

A simple example of a serializing mechanism is the use of exclusive locks. In this locking scheme, the server attempts to lock any object that is about to be used by any operation of a client's transaction. If a client requests access to an object that is already locked due to another client's transaction, the request is suspended and the client must wait until the object is unlocked.

Figure 16.14 illustrates the use of exclusive locks. It shows the same transactions as Figure 16.7, but with an extra column for each transaction showing the locking, waiting and unlocking. In this example, it is assumed that when transactions  $T$  and  $U$  start, the balances of the accounts  $A$ ,  $B$  and  $C$  are not yet locked. When transaction  $T$  is about to use account  $B$ , it is locked for  $T$ . When transaction  $U$  is about to use  $B$  it is still

Figure 16.14 Transactions  $T$  and  $U$  with exclusive locks

| Transaction $T$ :  |               | Transaction $U$ :  |                                 |
|--|---------------|--|---------------------------------|
| <i>balance</i> = <i>b.getBalance</i> ()<br><i>b.setBalance</i> ( <i>bal</i> *1.1)<br><i>a.withdraw</i> ( <i>bal</i> /10) |               | <i>balance</i> = <i>b.getBalance</i> ()<br><i>b.setBalance</i> ( <i>bal</i> *1.1)<br><i>c.withdraw</i> ( <i>bal</i> /10) |                                 |
| Operations   | Locks         | Operations   | Locks                           |
| <i>openTransaction</i>   |               | <i>openTransaction</i>   |                                 |
| <i>bal</i> = <i>b.getBalance</i> ()  | lock $B$      | <i>bal</i> = <i>b.getBalance</i> ()  | waits for $T$ 's<br>lock on $B$ |
| <i>b.setBalance</i> ( <i>bal</i> *1.1)   |               | <i>closeTransaction</i>  |                                 |
| <i>a.withdraw</i> ( <i>bal</i> /10)  | lock $A$      | ...  |                                 |
| <i>closeTransaction</i>  | unlock $A, B$ |  | lock $B$                        |
|  |               | <i>b.setBalance</i> ( <i>bal</i> *1.1)   |                                 |
|  |               | <i>c.withdraw</i> ( <i>bal</i> /10)  | lock $C$                        |
|  |               | <i>closeTransaction</i>  | unlock $B, C$                   |

locked for  $T$ , so transaction  $U$  waits. When transaction  $T$  is committed,  $B$  is unlocked, whereupon transaction  $U$  is resumed. The use of the lock on  $B$  effectively serializes the access to  $B$ . Note that if, for example,  $T$  released the lock on  $B$  between its *getBalance* and *setBalance* operations, transaction  $U$ 's *getBalance* operation on  $B$  could be interleaved between them.

Serial equivalence requires that all of a transaction's accesses to a particular object be serialized with respect to accesses by other transactions. All pairs of conflicting operations of two transactions should be executed in the same order. To ensure this, a transaction is not allowed any new locks after it has released a lock. The first phase of each transaction is a 'growing phase', during which new locks are acquired. In the second phase, the locks are released (a 'shrinking phase'). This is called *two-phase locking*.

We saw in Section 16.2.2 that because transactions may abort, strict executions are needed to prevent dirty reads and premature writes. Under a strict execution regime, a transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted. To enforce this rule, any locks applied during the progress of a transaction are held until the transaction commits or aborts. This is called *strict two-phase locking*. The presence of the locks prevents other transactions reading or writing the objects. When a transaction commits, to ensure recoverability, the locks must be held until all the objects it updated have been written to permanent storage.

A server generally contains a large number of objects, and a typical transaction accesses only a few of them and is unlikely to clash with other current transactions. The *granularity* with which concurrency control can be applied to objects is an important

issue, since the scope for concurrent access to objects in a server will be limited severely if concurrency control (for example, locks) can only be applied to all the objects at once. In our banking example, if locks were applied to all customer accounts at a branch, only one bank clerk could perform an online banking transaction at any time – hardly an acceptable constraint!

The portion of the objects to which access must be serialized should be as small as possible; that is, just that part involved in each operation requested by transactions. In our banking example, a branch holds a set of accounts, each of which has a balance. Each banking operation affects one or more account balances – *deposit* and *withdraw* affect one account balance, and *branchTotal* affects all of them.

The description of concurrency control schemes given below does not assume any particular granularity. We discuss concurrency control protocols that are applicable to objects whose operations can be modelled in terms of *read* and *write* operations on the objects. For the protocols to work correctly, it is essential that each *read* and *write* operation is atomic in its effects on objects.

Concurrency control protocols are designed to cope with *conflicts* between operations in different transactions on the same object. In this chapter, we use the notion of conflict between operations to explain the protocols. The conflict rules for *read* and *write* operations are given in Figure 16.9, which shows that pairs of *read* operations from different transactions on the same object do not conflict. Therefore, a simple exclusive lock that is used for both *read* and *write* operations reduces concurrency more than is necessary.

It is preferable to adopt a locking scheme that controls the access to each object so that there can be several concurrent transactions reading an object, or a single transaction writing an object, but not both. This is commonly referred to as a ‘many readers/single writer’ scheme. Two types of locks are used: *read locks* and *write locks*. Before a transaction’s *read* operation is performed, a read lock should be set on the object. Before a transaction’s *write* operation is performed, a write lock should be set on the object. Whenever it is impossible to set a lock immediately, the transaction (and the client) must wait until it is possible to do so – a client’s request is never rejected.

As pairs of *read* operations from different transactions do not conflict, an attempt to set a read lock on an object with a read lock is always successful. All the transactions reading the same object share its read lock – for this reason, read locks are sometimes called *shared locks*.

The operation conflict rules tell us that:

1. If a transaction *T* has already performed a *read* operation on a particular object, then a concurrent transaction *U* must not *write* that object until *T* commits or aborts.
2. If a transaction *T* has already performed a *write* operation on a particular object, then a concurrent transaction *U* must not *read* or *write* that object until *T* commits or aborts.

To enforce condition 1, a request for a write lock on an object is delayed by the presence of a read lock belonging to another transaction. To enforce condition 2, a request for either a read lock or a write lock on an object is delayed by the presence of a write lock belonging to another transaction.

Figure 16.15 Lock compatibility

| For one object          |              | Lock requested |              |
|-------------------------|--------------|----------------|--------------|
|                         |              | <i>read</i>    | <i>write</i> |
| <i>Lock already set</i> | <i>none</i>  | OK             | OK           |
|                         | <i>read</i>  | OK             | wait         |
|                         | <i>write</i> | wait           | wait         |

Figure 16.15 shows the compatibility of read locks and write locks on any particular object. The entries to the left of the first column in the table show the type of lock already set, if any. The entries above the first row show the type of lock requested. The entry in each cell shows the effect on a transaction that requests the type of lock given above when the object has been locked in another transaction with the type of lock on the left.

Inconsistent retrievals and lost updates are caused by conflicts between *read* operations in one transaction and *write* operations in another without the protection of a concurrency control scheme such as locking. Inconsistent retrievals are prevented by performing the retrieval transaction before or after the update transaction. If the retrieval transaction comes first, its read locks delay the update transaction. If it comes second, its request for read locks causes it to be delayed until the update transaction has completed.

Lost updates occur when two transactions read a value of an object and then use it to calculate a new value. Lost updates are prevented by making later transactions delay their reads until the earlier ones have completed. This is achieved by each transaction setting a read lock when it reads an object and then *promoting* it to a write lock when it writes the same object – when a subsequent transaction requires a read lock it will be delayed until any current transaction has completed.

A transaction with a read lock that is shared with other transactions cannot promote its read lock to a write lock, because the latter would conflict with the read locks held by the other transactions. Therefore, such a transaction must request a write lock and wait for the other read locks to be released.

Lock promotion refers to the conversion of a lock to a stronger lock – that is, a lock that is more exclusive. The lock compatibility table in Figure 16.15 shows the relative exclusivity of locks. The read lock allows other read locks, whereas the write lock does not. Neither allows other write locks. Therefore, a write lock is more exclusive than a read lock. Locks may be promoted because the result is a more exclusive lock. It is not safe to demote a lock held by a transaction before it commits, because the result will be more permissive than the previous one and may allow executions by other transactions that are inconsistent with serial equivalence.

The rules for the use of locks in a strict two-phase locking implementation are summarized in Figure 16.16. To ensure that these rules are adhered to, the client has no access to operations for locking or unlocking items of data. Locking is performed when the requests for *read* and *write* operations are about to be applied to the recoverable objects, and unlocking is performed by the *commit* or *abort* operations of the transaction coordinator.

Figure 16.16 Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
    - (a) If the object is not already locked, it is locked and the operation proceeds.
    - (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
    - (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
    - (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule b is used.)
  2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.
- 

For example, the CORBA Concurrency Control Service [OMG 2000b] can be used to apply concurrency control on behalf of transactions or to protect objects without using transactions. It provides a means of associating a collection of locks (called a *lockset*) with a resource such as a recoverable object. A lockset allows locks to be acquired or released. A lockset's *lock* method will acquire a lock or block until the lock is free; other methods allow locks to be promoted or released. Transactional locksets support the same methods as locksets, but their methods require transaction identifiers as arguments. We mentioned earlier that the CORBA transaction service tags all client requests in a transaction with the transaction identifier. This enables a suitable lock to be acquired before each of the recoverable objects is accessed during a transaction. The transaction coordinator is responsible for releasing the locks when a transaction commits or aborts.

The rules given in Figure 16.16 ensure strictness, because the locks are held until a transaction has either committed or aborted. However, it is not necessary to hold read locks to ensure strictness. Read locks need only be held until the request to commit or abort arrives.

**Lock implementation** • The granting of locks will be implemented by a separate object in the server that we call the *lock manager*. The lock manager holds a set of locks, for example in a hash table. Each lock is an instance of the class *Lock* and is associated with a particular object. The class *Lock* is shown in Figure 16.17. Each instance of *Lock* maintains the following information in its instance variables:

- the identifier of the locked object;
- the transaction identifiers of the transactions that currently hold the lock (shared locks can have several holders);
- a lock type.

Figure 16.17 Lock class

```

public class Lock {
    private Object object;      // the object being protected by the lock
    private Vector holders;     // the TIDs of current holders
    private LockType lockType; // the current type

    public synchronized void acquire(TransID trans, LockType aLockType ){
        while(/*another transaction holds the lock in conflicting mode*/) {
            try {
                wait();
            } catch ( InterruptedException e){/*...*/ }
        }
        if (holders.isEmpty()) { // no TIDs hold lock
            holders.addElement(trans);
            lockType = aLockType;
        } else if (/*another transaction holds the lock, share it*/ ) {
            if (/* this transaction not a holder*/) holders.addElement(trans);
        } else if (/* this transaction is a holder but needs a more exclusive lock*/)
            lockType.promote();
        }
    }

    public synchronized void release(TransID trans ){
        holders.removeElement(trans); // remove this holder
        // set locktype to none
        notifyAll();
    }
}

```

The methods of *Lock* are synchronized so that the threads attempting to acquire or release a lock will not interfere with one another. But, in addition, attempts to acquire the lock use the *wait* method whenever they have to wait for another thread to release it.

The *acquire* method carries out the rules given in Figure 16.15 and Figure 16.16. Its arguments specify a transaction identifier and the type of lock required by that transaction. It tests whether the request can be granted. If another transaction holds the lock in a conflicting mode, it invokes *wait*, which causes the caller's thread to be suspended until a corresponding *notify*. Note that the *wait* is enclosed in a *while*, because all waiters are notified and some of them may not be able to proceed. When, eventually, the condition is satisfied, the remainder of the method sets the lock appropriately:

- if no other transaction holds the lock, just add the given transaction to the holders and set the type;
- else if another transaction holds the lock, share it by adding the given transaction to the holders (unless it is already a holder);
- else if this transaction is a holder but is requesting a more exclusive lock, promote the lock.

Figure 16.18 *LockManager* class

```

public class LockManager {
    private Hashtable theLocks;

    public void setLock(Object object, TransID trans, LockType lockType){
        Lock foundLock;
        synchronized(this){
            // find the lock associated with object
            // if there isn't one, create it and add it to the hashtable
        }
        foundLock.acquire(trans, lockType);
    }

    // synchronize this one because we want to remove all entries
    public synchronized void unLock(TransID trans) {
        Enumeration e = theLocks.elements();
        while(e.hasMoreElements()){
            Lock aLock = (Lock)(e.nextElement());
            if(/* trans is a holder of this lock*/ ) aLock.release(trans);
        }
    }
}

```

The *release* method's arguments specify the transaction identifier of the transaction that is releasing the lock. It removes the transaction identifier from the holders, sets the lock type to *none* and calls *notifyAll*. The method notifies all waiting threads in case there are multiple transactions waiting to acquire read locks – all of them may be able to proceed.

The class *LockManager* is shown in Figure 16.18. All requests to set locks and to release them on behalf of transactions are sent to an instance of *LockManager*:

- The *setLock* method's arguments specify the object that the given transaction wants to lock and the type of lock. It finds a lock for that object in its hashtable or, if necessary, creates one. It then invokes the *acquire* method of that lock.
- The *unLock* method's argument specifies the transaction that is releasing its locks. It finds all of the locks in the hashtable that have the given transaction as a holder. For each one, it calls the *release* method.

**Some questions of policy:** Note that when several threads *wait* on the same locked item, the semantics of *wait* ensure that each transaction gets its turn. In the above program, the conflict rules allow the holders of a lock to be either multiple readers or one writer. The arrival of a request for a read lock is always granted unless the holder has a write lock.

The reader is invited to consider the following:

What is the consequence for *write* transactions in the presence of a steady trickle of requests for read locks? Think of an alternative implementation.



When the holder has a write lock, several readers and writers may be waiting. The reader should consider the effect of *notifyAll* and think of an alternative implementation. If a holder of a read lock tries to promote the lock when the lock is shared, it will be blocked. Is there any solution to this difficulty?

**Locking rules for nested transactions** • The aim of a locking scheme for nested transactions is to serialize access to objects so that:

1. Each set of nested transactions is a single entity that must be prevented from observing the partial effects of any other set of nested transactions.
2. Each transaction within a set of nested transactions must be prevented from observing the partial effects of the other transactions in the set.

The first rule is enforced by arranging that every lock that is acquired by a successful subtransaction is *inherited* by its parent when it completes. Inherited locks are also inherited by ancestors. Note that this form of inheritance passes from child to parent! The top-level transaction eventually inherits all of the locks that were acquired by successful subtransactions at any depth in a nested transaction. This ensures that the locks can be held until the top-level transaction has committed or aborted, which prevents members of different sets of nested transactions observing one another's partial effects.

The second rule is enforced as follows:

- Parent transactions are not allowed to run concurrently with their child transactions. If a parent transaction has a lock on an object, it *retains* the lock during the time that its child transaction is executing. This means that the child transaction temporarily acquires the lock from its parent for its duration.
- Subtransactions at the same level are allowed to run concurrently, so when they access the same objects, the locking scheme must serialize their access.

The following rules describe lock acquisition and release:

- For a subtransaction to acquire a read lock on an object, no other active transaction can have a write lock on that object, and the only retainers of a write lock are its ancestors.
- For a subtransaction to acquire a write lock on an object, no other active transaction can have a read or write lock on that object, and the only retainers of read and write locks on that object are its ancestors.
- When a subtransaction commits, its locks are inherited by its parent, allowing the parent to retain the locks in the same mode as the child.
- When a subtransaction aborts, its locks are discarded. If the parent already retains the locks, it can continue to do so.

Note that subtransactions at the same level that access the same object will take turns to acquire the locks retained by their parent. This ensures that their access to a common object is serialized.

As an example, suppose that subtransactions  $T_1$ ,  $T_2$  and  $T_{11}$  in Figure 16.13 all access a common object, which is not accessed by the top-level transaction  $T$ . Suppose that subtransaction  $T_1$  is the first to access the object and successfully acquires a lock,

Figure 16.19 Deadlock with write locks

| Transaction <i>T</i>   |   | Transaction <i>U</i>    |   |
|------------------------|---|-------------------------|---|
| Operations             | Locks                                     | Operations              | Locks                                     |
| <i>a.deposit(100);</i> | write lock <i>A</i>                       | <i>b.deposit(200)</i>   | write lock <i>B</i>                       |
| <i>b.withdraw(100)</i> |   |                         |   |
| ...                    | waits for <i>U</i> 's<br>lock on <i>B</i> | <i>a.withdraw(200);</i> | waits for <i>T</i> 's<br>lock on <i>A</i> |
| ...                    |   | ...                     |   |
| ...                    |   | ...                     |   |

which it passes on to  $T_{11}$  for the duration of its execution, getting it back when  $T_{11}$  completes. When  $T_1$  completes its execution, the top-level transaction  $T$  inherits the lock, which it retains until the set of nested transactions completes. The subtransaction  $T_2$  can acquire the lock from  $T$  for the duration of its execution.

16.4.1 Deadlocks

The use of locks can lead to deadlock. Consider the use of locks shown in Figure 16.19. Since the *deposit* and *withdraw* methods are atomic, we show them acquiring write locks – although in practice they read the balance and then write it. Each of them acquires a lock on one account and then gets blocked when it tries to access the account that the other one has locked. This is a deadlock situation – two transactions are waiting, and each is dependent on the other to release a lock so it can resume.

Deadlock is a particularly common situation when clients are involved in an interactive program, for a transaction in an interactive program may last for a long period of time. This can result in many objects being locked and remaining so, thus preventing other clients using them.

Note that the locking of subitems in structured objects can be useful for avoiding conflicts and possible deadlock situations. For example, a day in a diary could be structured as a set of timeslots, each of which can be locked independently for updating. Hierarchic locking schemes are useful if the application requires a different granularity of locking for different operations, see Section 16.4.2.

**Definition of deadlock** • Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock. A *wait-for graph* can be used to represent the waiting relationships between current transactions. In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions – there is an edge from node  $T$  to node  $U$  when transaction  $T$  is waiting for transaction  $U$  to release a lock. Figure 16.20 illustrates the wait-for graph corresponding to the deadlock situation illustrated in Figure 16.19. Recall that the deadlock arose because transactions  $T$  and  $U$  both attempted to acquire an object held by the other. Therefore  $T$  waits for  $U$  and  $U$  waits for  $T$ . The dependency between

Figure 16.20 The wait-for graph for Figure 16.19

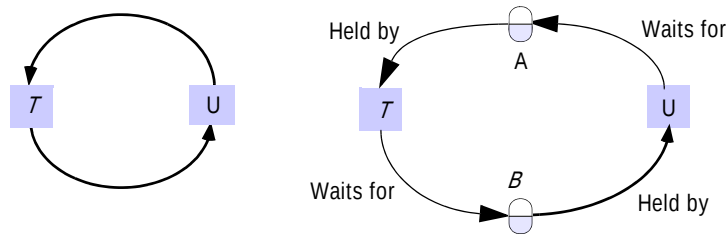
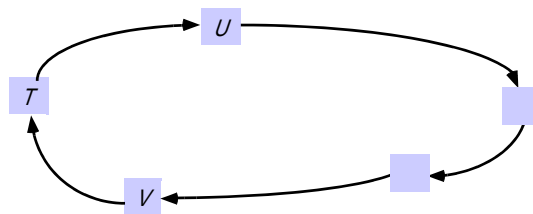


Figure 16.21 A cycle in a wait-for graph



transactions is indirect, via a dependency on objects. The diagram on the right shows the objects held by and waited for by transactions  $T$  and  $U$ . As each transaction can wait for only one object, the objects can be omitted from the wait-for graph – leaving the simple graph on the left.

Suppose that, as in Figure 16.21, a wait-for graph contains a cycle  $T \rightarrow U \rightarrow \dots \rightarrow V \rightarrow T$ . Each transaction is waiting for the next transaction in the cycle. All of these transactions are blocked waiting for locks. None of the locks can ever be released, and the transactions are deadlocked. If one of the transactions in a cycle is aborted, then its locks are released and that cycle is broken. For example, if transaction  $T$  in Figure 16.21 is aborted, it will release a lock on an object that  $V$  is waiting for – and  $V$  will no longer be waiting for  $T$ .

Now consider a scenario in which the three transactions  $T$ ,  $U$  and  $V$  share a read lock on an object  $C$ , and transaction  $W$  holds a write lock on object  $B$ , on which transaction  $V$  is waiting to obtain a lock (as shown on the right in Figure 16.22). The transactions  $T$  and  $W$  then request write locks on object  $C$ , and a deadlock situation arises in which  $T$  waits for  $U$  and  $V$ ,  $V$  waits for  $W$ , and  $W$  waits for  $T$ ,  $U$  and  $V$ , as shown on the left in Figure 16.22. This shows that although each transaction can wait for only one object at a time, it may be involved in several cycles. For example, transaction  $V$  is involved in two cycles:  $V \rightarrow W \rightarrow T \rightarrow V$  and  $V \rightarrow W \rightarrow V$ .

In this example, suppose that transaction  $V$  is aborted. This will release  $V$ 's lock on  $C$  and the two cycles involving  $V$  will be broken.

**Deadlock prevention** • One solution is to prevent deadlock. An apparently simple but not very good way to overcome the deadlock problem is to lock all of the objects used by a transaction when it starts. This would need to be done as a single atomic step so as



Figure 16.23 Resolution of the deadlock in Figure 16.19

| Transaction T  |                   | Transaction U           |               |
|--|-------------------|-------------------------|---------------|
| Operations   | Locks             | Operations              | Locks         |
| <i>a.deposit(100);</i>                                 | write lock A      |                         |               |
|  |                   | <i>b.deposit(200)</i>   | write lock B  |
| <i>b.withdraw(100)</i>                                 |                   |                         |               |
| ...  | waits for U's     | <i>a.withdraw(200);</i> | waits for T's |
|  | lock on B         | ...                     | lock on A     |
|  | (timeout elapses) | ...                     |               |
| T's lock on A becomes vulnerable,<br>unlock A, abort T |                   | <i>a.withdraw(200);</i> | write lock A  |
|  |                   |                         | unlock A, B   |

a lock is shared, several edges may be added. An edge  $T \rightarrow U$  is deleted whenever  $U$  releases a lock that  $T$  is waiting for and allows  $T$  to proceed. See Exercise 16.14 for a more detailed discussion of the implementation of deadlock detection. If a transaction shares a lock, the lock is not released, but the edges leading to a particular transaction are removed.

The presence of cycles may be checked each time an edge is added, or less frequently to avoid unnecessary overhead. When a deadlock is detected, one of the transactions in the cycle must be chosen and then be aborted. The corresponding node and the edges involving it must be removed from the wait-for graph. This will happen when the aborted transaction has its locks removed.

The choice of the transaction to abort is not simple. Some factors that may be taken into account are the age of the transaction and the number of cycles in which it is involved.

**Timeouts** • Lock timeouts are a method for resolution of deadlocks that is commonly used. Each lock is given a limited period in which it is invulnerable. After this time, a lock becomes vulnerable. Provided that no other transaction is competing for the object that is locked, an object with a vulnerable lock remains locked. However, if any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken (that is, the object is unlocked) and the waiting transaction resumes. The transaction whose lock has been broken is normally aborted.

There are many problems with the use of timeouts as a remedy for deadlocks: the worst problem is that transactions are sometimes aborted due to their locks becoming vulnerable when other transactions are waiting for them, but there is actually no deadlock. In an overloaded system, the number of transactions timing out will increase, and transactions taking a long time can be penalized. In addition, it is hard to decide on an appropriate length for a timeout. In contrast, if deadlock detection is used,

Figure 16.24 Lock compatibility (*read*, *write* and *commit* locks)

| For one object          |               | Lock to be set |              |               |
|-------------------------|---------------|----------------|--------------|---------------|
|                         |               | <i>read</i>    | <i>write</i> | <i>commit</i> |
| <i>Lock already set</i> | <i>none</i>   | OK             | OK           | OK            |
|                         | <i>read</i>   | OK             | OK           | wait          |
|                         | <i>write</i>  | OK             | wait         | —             |
|                         | <i>commit</i> | wait           | wait         | —             |

transactions are aborted because deadlocks have occurred and a choice can be made as to which transaction to abort.

Using lock timeouts, we can resolve the deadlock in Figure 16.19 as shown in Figure 16.23, in which the write lock for *T* on *A* becomes vulnerable after its timeout period. Transaction *U* is waiting to acquire a write lock on *A*. Therefore, *T* is aborted and it releases its lock on *A*, allowing *U* to resume and complete the transaction.

When transactions access objects located in several different servers, the possibility of distributed deadlocks arises. In a distributed deadlock, the wait-for graph can involve objects at multiple locations. We return to this subject in Section 17.5.

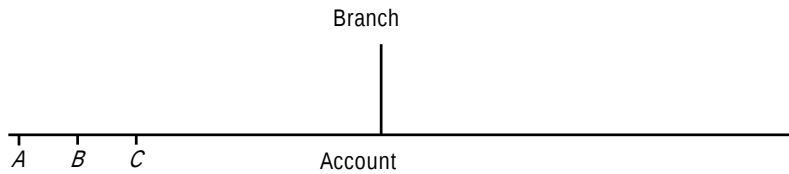
### 16.4.2 Increasing concurrency in locking schemes

Even when locking rules are based on the conflicts between *read* and *write* operations and the granularity at which they are applied is as small as possible, there is still some scope for increasing concurrency. We discuss two approaches that have been used to deal with this issue. In the first approach (two-version locking), the setting of exclusive locks is delayed until a transaction commits. In the second approach (hierarchic locks), mixed-granularity locks are used.

**Two-version locking** • This is an optimistic scheme that allows one transaction to write tentative versions of objects while other transactions read from the committed versions of the same objects. *read* operations only wait if another transaction is currently committing the same object. This scheme allows more concurrency than read-write locks, but writing transactions risk waiting or even rejection when they attempt to commit. Transactions cannot commit their *write* operations immediately if other uncompleted transactions have read the same objects. Therefore, transactions that request to commit in such a situation are made to wait until the reading transactions have completed. Deadlocks may occur when transactions are waiting to commit. Therefore, transactions may need to be aborted when they are waiting to commit, to resolve deadlocks.

This variation on strict two-phase locking uses three types of lock: a read lock, a write lock and a commit lock. Before a transaction’s *read* operation is performed, a read lock must be set on the object – the attempt to set a read lock is successful unless the object has a commit lock, in which case the transaction waits. Before a transaction’s

Figure 16.25 Lock hierarchy for the banking example



*write* operation is performed, a write lock must be set on the object – the attempt to set a write lock is successful unless the object has a write lock or a commit lock, in which case the transaction waits.

When the transaction coordinator receives a request to commit a transaction, it attempts to convert all that transaction's write locks to commit locks. If any of the objects have outstanding read locks, the transaction must wait until the transactions that set these locks have completed and the locks are released. The compatibility of read, write and commit locks is shown in Figure 16.24.

There are two main differences in performance between the two-version locking scheme and an ordinary read-write locking scheme. On the one hand, *read* operations in the two-version locking scheme are delayed only while the transactions are being committed, rather than during the entire execution of transactions – in most cases, the commit protocol takes only a small fraction of the time required to perform an entire transaction. On the other hand, *read* operations of one transaction can cause delays in committing other transactions.

**Hierarchic locks** • In some applications, the granularity suitable for one operation is not appropriate for another operation. In our banking example, the majority of the operations require locking at the granularity of an account. The *branchTotal* operation is different – it reads the values of all the account balances and would appear to require a read lock on all of them. To reduce locking overhead, it would be useful to allow locks of mixed granularity to coexist.

Gray [1978] proposed the use of a hierarchy of locks with different granularities. At each level, the setting of a parent lock has the same effect as setting all the equivalent child locks. This economizes on the number of locks to be set. In our banking example, the branch is the parent and the accounts are children (see Figure 16.25).

Mixed-granularity locks could be useful in a diary system in which the data could be structured with the diary for a week being composed of a page for each day and the

Figure 16.26 Lock hierarchy for a diary

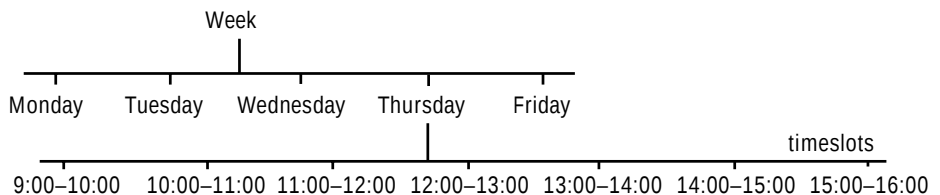


Figure 16.27 Lock compatibility table for hierarchic locks

| For one object   |         | Lock to be set |       |        |         |
|------------------|---------|----------------|-------|--------|---------|
|                  |         | read           | write | I-read | I-write |
| Lock already set | none    | OK             | OK    | OK     | OK      |
|                  | read    | OK             | wait  | OK     | wait    |
|                  | write   | wait           | wait  | wait   | wait    |
|                  | I-read  | OK             | wait  | OK     | OK      |
|                  | I-write | wait           | wait  | OK     | OK      |

latter subdivided further into a slot for each hour of the day, as shown in Figure 16.26. The operation to view a week would cause a read lock to be set at the top of this hierarchy, whereas the operation to enter an appointment would cause a write lock to be set on a given time slot. The effect of a read lock on a week would be to prevent write operations on any of the substructures – for example, the time slots for each day in that week.

In Gray’s scheme, each node in the hierarchy can be locked, giving the owner of the lock explicit access to the node and giving implicit access to its children. In our example, in Figure 16.25 a read-write lock on the branch implicitly read-write locks all the accounts. Before a child node is granted a read-write lock, an intention to read-write lock is set on the parent node and its ancestors (if any). The intention lock is compatible with other intention locks but conflicts with read and write locks according to the usual rules. Figure 16.27 gives the compatibility table for hierarchic locks. Gray also proposed a third type of intention lock – one that combines the properties of a read lock with an intention to write lock.

In our banking example, the *branchTotal* operation requests a read lock on the branch, which implicitly sets read locks on all the accounts. A *deposit* operation needs to set a write lock on a balance, but first it attempts to set an intention to write lock on the branch. These rules prevent these operations running concurrently.

Hierarchic locks have the advantage of reducing the number of locks when mixed-granularity locking is required. The compatibility tables and the rules for promoting locks are more complex.

The mixed granularity of locks could allow each transaction to lock a portion whose size is chosen according to its needs. A long transaction that accesses many objects could lock the whole collection, whereas a short transaction can lock at finer granularity.

The CORBA Concurrency Control Service supports variable-granularity locking with intention to read and intention to write lock types. These can be used as described above to take advantage the opportunity to apply locks at differing granularities in hierarchically structured data.



## 16.5 Optimistic concurrency control

Kung and Robinson [1981] identified a number of inherent disadvantages of locking and proposed an alternative optimistic approach to the serialization of transactions that avoids these drawbacks. We can summarize the drawbacks of locking:

- Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data. Even read-only transactions (queries), which cannot possibly affect the integrity of the data, must, in general, use locking in order to guarantee that the data being read is not modified by other transactions at the same time. But locking may be necessary only in the worst case.

For example, consider two client processes that are concurrently incrementing the values of  $n$  objects. If the client programs start at the same time and run for about the same amount of time, accessing the objects in two unrelated sequences and using a separate transaction to access and increment each item, the chances that the two programs will attempt to access the same object at the same time are just  $1/n$  on average, so locking is really needed only once in every  $n$  transactions.

- The use of locks can result in deadlock. Deadlock prevention reduces concurrency severely, and therefore deadlock situations must be resolved either by the use of timeouts or by deadlock detection. Neither of these is wholly satisfactory for use in interactive programs.
- To avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce significantly the potential for concurrency.

The alternative approach proposed by Kung and Robinson is ‘optimistic’ because it is based on the observation that, in most applications, the likelihood of two clients’ transactions accessing the same object is low. Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client completes its task and issues a *closeTransaction* request. When a conflict arises, some transaction is generally aborted and will need to be restarted by the client. Each transaction has the following phases:

*Working phase:* During the working phase, each transaction has a tentative version of each of the objects that it updates. This is a copy of the most recently committed version of the object. The use of tentative versions allows the transaction to abort (with no effect on the objects), either during the working phase or if it fails validation due to other conflicting transactions. *read* operations are performed immediately – if a tentative version for that transaction already exists, a *read* operation accesses it; otherwise, it accesses the most recently committed value of the object. *write* operations record the new values of the objects as tentative values (which are invisible to other transactions). When there are several concurrent transactions, several different tentative values of the same object may coexist. In addition, two records are kept of the objects accessed within a transaction: a *read set* containing the objects read by the transaction and a *write set* containing the objects written by the transaction. Note that as all *read* operations are performed on committed versions of the objects (or copies of them), dirty reads cannot occur.

*Validation phase:* When the *closeTransaction* request is received, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transactions on the same objects. If the validation is successful, then the transaction can commit. If the validation fails, then some form of conflict resolution must be used and either the current transaction or, in some cases, those with which it conflicts will need to be aborted.

*Update phase:* If a transaction is validated, all of the changes recorded in its tentative versions are made permanent. Read-only transactions can commit immediately after passing validation. Write transactions are ready to commit once the tentative versions of the objects have been recorded in permanent storage.

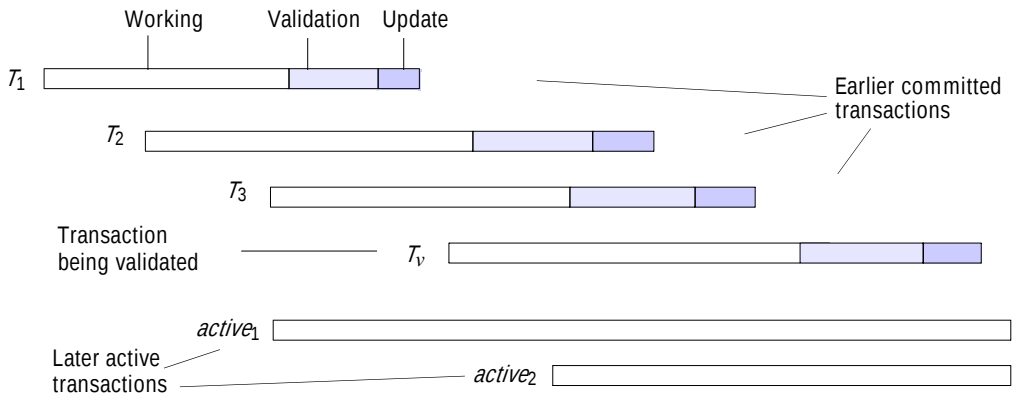
**Validation of transactions** • Validation uses the read-write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other *overlapping* transactions – that is, any transactions that had not yet committed at the time this transaction started. To assist in performing validation, each transaction is assigned a transaction number when it enters the validation phase (that is, when the client issues a *closeTransaction*). If the transaction is validated and completes successfully, it retains this number; if it fails the validation checks and is aborted, or if the transaction is read only, the number is released for reassignment. Transaction numbers are integers assigned in ascending sequence; the number of a transaction therefore defines its position in time – a transaction always finishes its working phase after all transactions with lower numbers. That is, a transaction with the number  $T_i$  always precedes a transaction with the number  $T_j$  if  $i < j$ . (If the transaction number were to be assigned at the beginning of the working phase, then a transaction that reached the end of the working phase before one with a lower number would have to wait until the earlier one had completed before it could be validated.)

The validation test on transaction  $T_v$  is based on conflicts between operations in pairs of transactions  $T_i$  and  $T_v$ . For a transaction  $T_v$  to be serializable with respect to an overlapping transaction  $T_i$ , their operations must conform to the following rules:

| $T_v$        | $T_i$        | Rule   |
|--------------|--------------|--|
| <i>write</i> | <i>read</i>  | 1. $T_i$ must not read objects written by $T_v$ .  |
| <i>read</i>  | <i>write</i> | 2. $T_v$ must not read objects written by $T_i$ .  |
| <i>write</i> | <i>write</i> | 3. $T_i$ must not write objects written by $T_v$ and $T_v$ must not write objects written by $T_i$ . |

As the validation and update phases of a transaction are generally short in duration compared with the working phase, a simplification can be achieved by making the rule that only one transaction may be in the validation and update phase at one time. When no two transactions may overlap in the update phase, rule 3 is satisfied. Note that this restriction on *write* operations, together with the fact that no dirty reads can occur, produces strict executions. To prevent overlapping, the entire validation and update phases can be implemented as a critical section so that only one client at a time can execute it. In order to increase concurrency, part of the validation and updating may be

Figure 16.28 Validation of transactions



implemented outside the critical section, but it is essential that the assignment of transaction numbers is performed sequentially. We note that at any instant, the current transaction number is like a pseudo-clock that ticks whenever a transaction completes successfully.

The validation of a transaction must ensure that rules 1 and 2 are obeyed by testing for overlaps between the objects of pairs of transactions  $T_v$  and  $T_i$ . There are two forms of validation – backward and forward [Härder 1984]. Backward validation checks the transaction undergoing validation with other preceding overlapping transactions – those that entered the validation phase before it. Forward validation checks the transaction undergoing validation with other later transactions, which are still active.

**Backward validation** • As all the *read* operations of earlier overlapping transactions were performed before the validation of  $T_v$  started, they cannot be affected by the *writes* of the current transaction (and rule 1 is satisfied). The validation of transaction  $T_v$  checks whether its read set (the objects affected by the *read* operations of  $T_v$ ) overlaps with any of the write sets of earlier overlapping transactions,  $T_i$  (rule 2). If there is any overlap, the validation fails.

Let  $startTn$  be the biggest transaction number assigned (to some other committed transaction) at the time when transaction  $T_v$  started its working phase and  $finishTn$  be the biggest transaction number assigned at the time when  $T_v$  entered the validation phase. The following program describes the algorithm for the validation of  $T_v$ :

```

boolean valid = true;
for (int  $T_i = startTn + 1$ ;  $T_i \leq finishTn$ ;  $T_i++$ ) {
    if (read set of  $T_v$  intersects write set of  $T_i$ ) valid = false;
}

```

Figure 16.28 shows overlapping transactions that might be considered in the validation of a transaction  $T_v$ . Time increases from left to right. The earlier committed transactions are  $T_1$ ,  $T_2$  and  $T_3$ .  $T_1$  committed before  $T_v$  started.  $T_2$  and  $T_3$  committed before  $T_v$  finished its working phase.  $startTn + 1 = T_2$  and  $finishTn = T_3$ . In backward validation, the read set of  $T_v$  must be compared with the write sets of  $T_2$  and  $T_3$ .

In backward validation, the read set of the transaction being validated is compared with the write sets of other transactions that have already committed. Therefore, the only way to resolve any conflicts is to abort the transaction that is undergoing validation.

In backward validation, transactions that have no *read* operations (only *write* operations) need not be checked.

Optimistic concurrency control with backward validation requires that the write sets of old committed versions of objects corresponding to recently committed transactions are retained until there are no unvalidated overlapping transactions with which they might conflict. Whenever a transaction is successfully validated, its transaction number, *startTn* and write set are recorded in a preceding transactions list that is maintained by the transaction service. Note that this list is ordered by transaction number. In an environment with long transactions, the retention of old write sets of objects may be a problem. For example, in Figure 16.28 the write sets of  $T_1$ ,  $T_2$ ,  $T_3$  and  $T_v$  must be retained until the active transaction  $active_1$  completes. Note that although the active transactions have transaction identifiers, they do not yet have transaction numbers.

**Forward validation** • In forward validation of the transaction  $T_v$ , the write set of  $T_v$  is compared with the read sets of all overlapping active transactions – those that are still in their working phase (rule 1). Rule 2 is automatically fulfilled because the active transactions do not write until after  $T_v$  has completed. Let the active transactions have (consecutive) transaction identifiers  $active_1$  to  $active_N$ . The following program describes the algorithm for the forward validation of  $T_v$ :

```
boolean valid = true;
for (int  $T_{id} = active_1$ ;  $T_{id} \leq active_N$ ;  $T_{id}++$ ) {
    if (write set of  $T_v$  intersects read set of  $T_{id}$ ) valid = false;
}
```

In Figure 16.28, the write set of transaction  $T_v$  must be compared with the read sets of the transactions with identifiers  $active_1$  and  $active_2$ . (Forward validation should allow for the fact that read sets of active transactions may change during validation and writing.) As the read sets of the transaction being validated are not included in the check, read-only transactions always pass the validation check. As the transactions being compared with the validating transaction are still active, we have a choice of whether to abort the validating transaction or to pursue some alternative way of resolving the conflict. Härder [1984] suggests several alternative strategies:

- Defer the validation until a later time when the conflicting transactions have finished. However, there is no guarantee that the transaction being validated will fare any better in the future. There is always the chance that further conflicting active transactions may start before the validation is achieved.
- Abort all the conflicting active transactions and commit the transaction being validated.
- Abort the transaction being validated. This is the simplest strategy but has the disadvantage that future conflicting transactions may be going to abort, in which case the transaction under validation has aborted unnecessarily.

**Comparison of forward and backward validation** • We have already seen that forward validation allows flexibility in the resolution of conflicts, whereas backward validation allows only one choice – to abort the transaction being validated. In general, the read sets of transactions are much larger than the write sets. Therefore, backward validation compares a possibly large read set against the old write sets, whereas forward validation checks a small write set against the read sets of active transactions. We see that backward validation has the overhead of storing old write sets until they are no longer needed. On the other hand, forward validation has to allow for new transactions starting during the validation process.

**Starvation** • When a transaction is aborted, it will normally be restarted by the client program. But in schemes that rely on aborting and restarting transactions, there is no guarantee that a particular transaction will ever pass the validation checks, for it may come into conflict with other transactions for the use of objects each time it is restarted. The prevention of a transaction ever being able to commit is called starvation.

Occurrences of starvation are likely to be rare, but a server that uses optimistic concurrency control must ensure that a client does not have its transaction aborted repeatedly. Kung and Robinson suggest that this could be done if the server detects a transaction that has been aborted several times. They suggest that when the server detects such a transaction it should be given exclusive access by the use of a critical section protected by a semaphore.

## 16.6 Timestamp ordering

In concurrency control schemes based on timestamp ordering, each operation in a transaction is validated when it is carried out. If the operation cannot be validated, the transaction is aborted immediately and can then be restarted by the client. Each transaction is assigned a unique timestamp value when it starts. The timestamp defines its position in the time sequence of transactions. Requests from transactions can be totally ordered according to their timestamps. The basic timestamp ordering rule is based on operation conflicts and is very simple:

A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transaction.

This rule assumes that there is only one version of each object and restricts access to one transaction at a time. If each transaction has its own tentative version of each object it accesses, then multiple concurrent transactions can access the same object. The timestamp ordering rule is refined to ensure that each transaction accesses a consistent set of versions of the objects. It must also ensure that the tentative versions of each object are committed in the order determined by the timestamps of the transactions that made them. This is achieved by transactions waiting, when necessary, for earlier transactions to complete their writes. The *write* operations may be performed after the *closeTransaction* operation has returned, without making the client wait. But the client must wait when *read* operations need to wait for earlier transactions to finish. This

Figure 16.29 Operation conflicts for timestamp ordering

| Rule | $T_c$ | $T_i$ |  |
|------|-------|-------|--|
| 1.   | write | read  | $T_c$ must not <i>write</i> an object that has been <i>read</i> by any $T_i$ where $T_i > T_c$ .<br>This requires that $T_c \geq$ the maximum read timestamp of the object.    |
| 2.   | write | write | $T_c$ must not <i>write</i> an object that has been <i>written</i> by any $T_i$ where $T_i > T_c$ .<br>This requires that $T_c >$ the write timestamp of the committed object. |
| 3.   | read  | write | $T_c$ must not <i>read</i> an object that has been <i>written</i> by any $T_i$ where $T_i > T_c$ .<br>This requires that $T_c >$ the write timestamp of the committed object.  |

cannot lead to deadlock, since transactions only wait for earlier ones (and no cycle could occur in the wait-for graph).

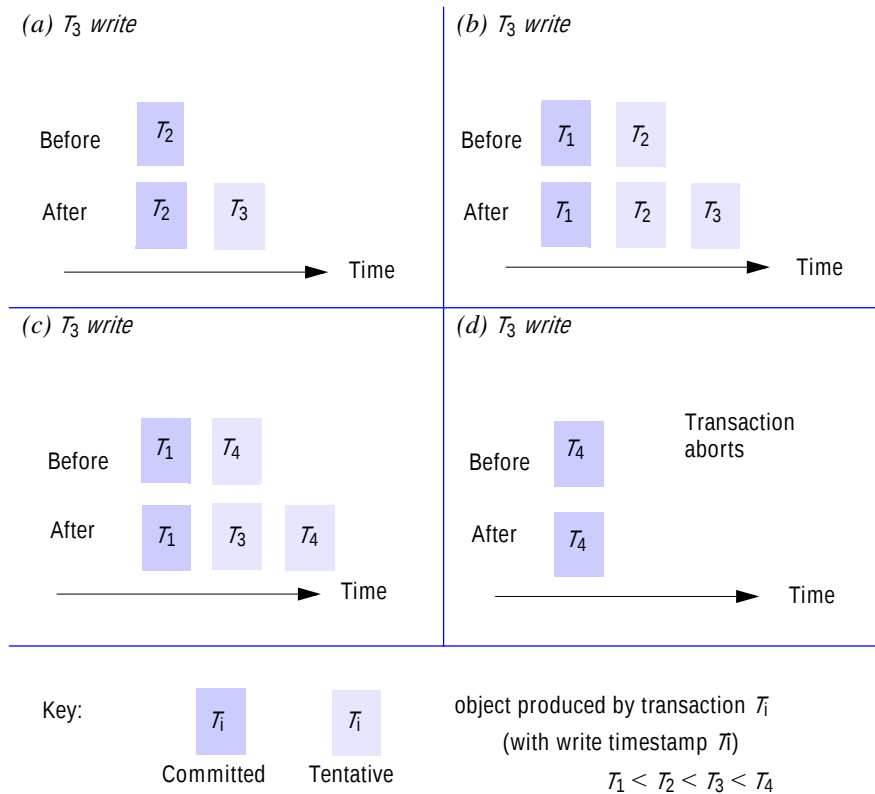
Timestamps may be assigned from the server's clock or, as in the previous section, a 'pseudo-time' may be based on a counter that is incremented whenever a timestamp value is issued. We defer until Chapter 17 the problem of generating timestamps when the transaction service is distributed and several servers are involved in a transaction.

We will now describe a form of timestamp-based concurrency control following the methods adopted in the SDD-1 system [Bernstein *et al.* 1980] and described by Ceri and Pelagatti [1985].

As usual, the *write* operations are recorded in tentative versions of objects and are invisible to other transactions until a *closeTransaction* request is issued and the transaction is committed. Every object has a write timestamp and a set of tentative versions, each of which has a write timestamp associated with it; each object also has a set of read timestamps. The write timestamp of the (committed) object is earlier than that of any of its tentative versions, and the set of read timestamps can be represented by its maximum member. Whenever a transaction's *write* operation on an object is accepted, the server creates a new tentative version of the object with its write timestamp set to the transaction timestamp. A transaction's *read* operation is directed to the version with the maximum write timestamp less than the transaction timestamp. Whenever a transaction's *read* operation on an object is accepted, the timestamp of the transaction is added to its set of read timestamps. When a transaction is committed, the values of the tentative versions become the values of the objects, and the timestamps of the tentative versions become the timestamps of the corresponding objects.

In timestamp ordering, each request by a transaction for a *read* or *write* operation on an object is checked to see whether it conforms to the operation conflict rules. A request by the current transaction  $T_c$  can conflict with previous operations done by other transactions,  $T_i$ , whose timestamps indicate that they should be later than  $T_c$ . These rules are shown in Figure 16.29, in which  $T_i > T_c$  means  $T_i$  is later than  $T_c$  and  $T_i < T_c$  means  $T_i$  is earlier than  $T_c$ .

Figure 16.30 Write operations and timestamps



**Timestamp ordering write rule:** By combining rules 1 and 2 we get the following rule for deciding whether to accept a *write* operation requested by transaction  $T_c$  on object  $D$ :

if ( $T_c \geq$  maximum read timestamp on  $D$  &&  
 $T_c >$  write timestamp on committed version of  $D$ )  
    perform *write* operation on tentative version of  $D$  with write timestamp  $T_c$   
else /\* write is too late \*/  
    Abort transaction  $T_c$

If a tentative version with write timestamp  $T_c$  already exists, the *write* operation is addressed to it; otherwise, a new tentative version is created and given write timestamp  $T_c$ . Note that any *write* that ‘arrives too late’ is aborted – it is too late in the sense that a transaction with a later timestamp has already read or written the object.

Figure 16.30 illustrates the action of a *write* operation by transaction  $T_3$  in cases where  $T_3 \geq$  maximum read timestamp on the object (the read timestamps are not shown). In cases (a) to (c),  $T_3 >$  write timestamp on the committed version of the object and a tentative version with write timestamp  $T_3$  is inserted at the appropriate place in the list of tentative versions ordered by their transaction timestamps. In case (d),  $T_3 <$  write timestamp on the committed version of the object and the transaction is aborted.

**Timestamp ordering read rule:** By using rule 3 we arrive at the following rule for deciding whether to accept immediately, to wait or to reject a *read* operation requested by transaction  $T_c$  on object  $D$ :

```

if (  $T_c >$  write timestamp on committed version of  $D$  ) {
    let  $D_{\text{selected}}$  be the version of  $D$  with the maximum write timestamp  $\delta T_c$ 
    if ( $D_{\text{selected}}$  is committed)
        perform read operation on the version  $D_{\text{selected}}$ 
    else
        wait until the transaction that made version  $D_{\text{selected}}$  commits or aborts
        then reapply the read rule
} else
    Abort transaction  $T_c$ 

```

Note:

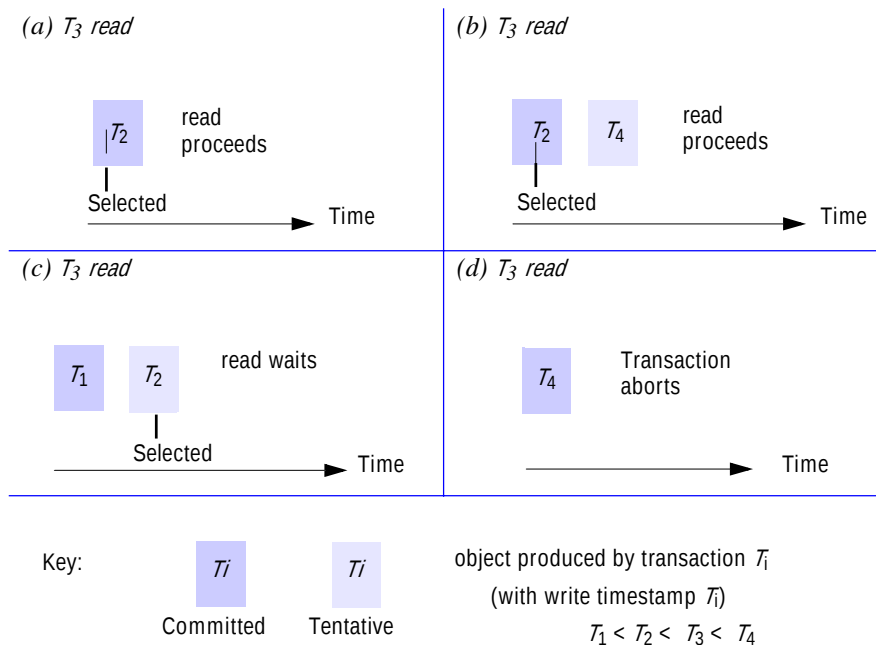
- If transaction  $T_c$  has already written its own version of the object, this will be used.
- A *read* operation that arrives too early waits for the earlier transaction to complete. If the earlier transaction commits, then  $T_c$  will read from its committed version. If it aborts, then  $T_c$  will repeat the read rule (and select the previous version). This rule prevents dirty reads.
- A *read* operation that ‘arrives too late’ is aborted – it is too late in the sense that a transaction with a later timestamp has already written the object.

Figure 16.31 illustrates the timestamp ordering read rule. It includes four cases labelled (a) to (d), each of which illustrates the action of a *read* operation by transaction  $T_3$ . In each case, a version whose write timestamp is less than or equal to  $T_3$  is selected. If such a version exists, it is indicated with a line. In cases (a) and (b) the *read* operation is directed to a committed version – in (a) it is the only version, whereas in (b) there is a tentative version belonging to a later transaction. In case (c) the *read* operation is directed to a tentative version and must wait until the transaction that made it commits or aborts. In case (d) there is no suitable version to read and transaction  $T_3$  is aborted.

When a coordinator receives a request to commit a transaction, it will always be able to do so because all the operations of transactions are checked for consistency with those of earlier transactions before being carried out. The committed versions of each object must be created in timestamp order. Therefore, a coordinator sometimes needs to wait for earlier transactions to complete before writing all the committed versions of the objects accessed by a particular transaction, but there is no need for the client to wait. In order to make a transaction recoverable after a server crash, the tentative versions of objects and the fact that the transaction has committed must be written to permanent storage before acknowledging the client’s request to commit the transaction.

Note that this timestamp ordering algorithm is a strict one – it ensures strict executions of transactions (see Section 16.2). The timestamp ordering read rule delays a transaction’s *read* operation on any object until all transactions that had previously written that object have committed or aborted. The arrangement to commit versions in order ensures that the execution of a transaction’s *write* operation on any object is delayed until all transactions that had previously written that object have committed or aborted.



Figure 16.31 *Read operations and timestamps*

In Figure 16.32, we return to our illustration concerning the two concurrent banking transactions  $T$  and  $U$  introduced in Figure 16.7. The columns headed  $A$ ,  $B$  and  $C$  refer to information about accounts with those names. Each account has an entry RTS that records the maximum read timestamp and an entry WTS that records the write timestamp of each version – with timestamps of committed versions in bold. Initially, all accounts have committed versions written by transaction  $S$ , and the set of read timestamps is empty. We assume  $S < T < U$ . The example shows that when transaction  $U$  is ready to get the balance of  $B$  it will wait for  $T$  to complete so that it can read the value set by  $T$  if it commits.

The timestamp method just described does avoid deadlocks, but it is quite likely to cause restarts. A modification known as the ‘ignore obsolete write’ rule is an improvement. This is a modification to the timestamp ordering write rule:

If a write is too late it can be ignored instead of aborting the transaction, because if it had arrived in time its effects would have been overwritten anyway. However, if another transaction has read the object, the transaction with the late write fails due to the read timestamp on the item.

**Multiversion timestamp ordering** • In this section, we have shown how the concurrency provided by basic timestamp ordering is improved by allowing each transaction to write its own tentative versions of objects. In multiversion timestamp ordering, which was introduced by Reed [1983], a list of old committed versions as well as tentative versions is kept for each object. This list represents the history of the values

Figure 16.32 Timestamps in transactions  $T$  and  $U$

|                         |                         | Timestamps and versions of objects |        |        |         |        |        |
|-------------------------|-------------------------|------------------------------------|--------|--------|---------|--------|--------|
| $T$                     | $U$                     | $A$                                |        | $B$    |         | $C$    |        |
|                         |                         | $RTS$                              | $WTS$  | $RTS$  | $WTS$   | $RTS$  | $WTS$  |
|                         |                         | $\{\}$                             | $S$    | $\{\}$ | $S$     | $\{\}$ | $S$    |
| $openTransaction$       |                         |                                    |        |        |         |        |        |
| $bal = b.getBalance()$  |                         |                                    |        |        | $\{T\}$ |        |        |
|                         | $openTransaction$       |                                    |        |        |         |        |        |
| $b.setBalance(bal*1.1)$ |                         |                                    |        |        |         | $S, T$ |        |
|                         | $bal = b.getBalance()$  |                                    |        |        |         |        |        |
|                         | $wait\ for\ T$          |                                    |        |        |         |        |        |
| $a.withdraw(bal/10)$    | $\dots$                 |                                    | $S, T$ |        |         |        |        |
| $commit$                | $\dots$                 |                                    | $T$    |        | $T$     |        |        |
|                         | $bal = b.getBalance()$  |                                    |        |        | $\{U\}$ |        |        |
|                         | $b.setBalance(bal*1.1)$ |                                    |        |        |         | $T, U$ |        |
|                         | $c.withdraw(bal/10)$    |                                    |        |        |         |        | $S, U$ |

of the object. The benefit of using multiple versions is that *read* operations that arrive too late need not be rejected.

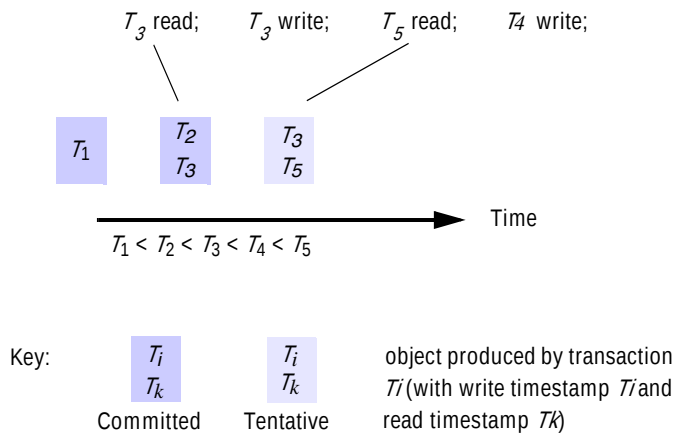
Each version has a read timestamp recording the largest timestamp of any transaction that has read from it in addition to a write timestamp. As before, whenever a *write* operation is accepted, it is directed to a tentative version with the write timestamp of the transaction. Whenever a *read* operation is carried out, it is directed to the version with the largest write timestamp less than the transaction timestamp. If the transaction timestamp is larger than the read timestamp of the version being used, the read timestamp of the version is set to the transaction timestamp.

When a read arrives late, it can be allowed to read from an old committed version, so there is no need to abort late *read* operations. In multiversion timestamp ordering, *read* operations are always permitted, although they may have to *wait* for earlier transactions to complete (either commit or abort), which ensures that executions are recoverable. See Exercise 16.22 for a discussion of the possibility of cascading aborts. This deals with rule 3 in the conflict rules for timestamp ordering.

There is no conflict between *write* operations of different transactions, because each transaction writes its own committed version of the objects it accesses. This removes rule 2 in the conflict rules for timestamp ordering, leaving us with:

Rule 1.  $T_c$  must not *write* objects that have been *read* by any  $T_i$  where  $T_i > T_c$ .

This rule will be broken if there is any version of the object with read timestamp  $> T_c$ , but only if this version has a write timestamp less than or equal to  $T_c$ . (This write cannot have any effect on later versions.)

Figure 16.33 Late *write* operation would invalidate a *read*

**Multiversion timestamp ordering write rule:** As any potentially conflicting *read* operation will have been directed to the most recent version of an object, the server inspects the version  $D_{maxEarlier}$  with the maximum write timestamp less than or equal to  $T_c$ . We have the following rule for performing a *write* operation requested by transaction  $T_c$  on object  $D$ :

if (read timestamp of  $D_{maxEarlier} \leq T_c$ )  
     perform *write* operation on a tentative version of  $D$  with write timestamp  $T_c$   
 else abort transaction  $T_c$

Figure 16.33 illustrates an example where a *write* is rejected. The object already has committed versions with write timestamps  $T_1$  and  $T_2$ . The object receives the following sequence of requests for operations on the object:

$T_3$  read;  $T_3$  write;  $T_5$  read;  $T_4$  write.

1.  $T_3$  requests a *read* operation, which puts a read timestamp  $T_3$  on  $T_2$ 's version.
2.  $T_3$  requests a *write* operation, which makes a new tentative version with write timestamp  $T_3$ .
3.  $T_5$  requests a *read* operation, which uses the version with write timestamp  $T_3$  (the highest timestamp that is less than  $T_5$ ).
4.  $T_4$  requests a *write* operation, which is rejected because the read timestamp  $T_5$  of the version with write timestamp  $T_3$  is bigger than  $T_4$ . (If it were permitted, the write timestamp of the new version would be  $T_4$ . If such a version were allowed, then it would invalidate  $T_5$ 's *read* operation, which should have used the version with timestamp  $T_4$ .)

When a transaction is aborted, all the versions that it created are deleted. When a transaction is committed, all the versions that it created are retained, but to control the use of storage space, old versions must be deleted from time to time. Although it has the overhead of storage space, multiversion timestamp ordering does allow considerable concurrency, does not suffer from deadlocks and always permits *read* operations. For further information about multiversion timestamp ordering, see Bernstein *et al.* [1987].

## 16.7 Comparison of methods for concurrency control

---

We have described three separate methods for controlling concurrent access to shared data: strict two-phase locking, optimistic methods and timestamp ordering. All of the methods carry some overheads in the time and space they require, and they all limit to some extent the potential for concurrent operation.

The timestamp ordering method is similar to two-phase locking in that both use pessimistic approaches in which conflicts between transactions are detected as each object is accessed. On the one hand, timestamp ordering decides the serialization order statically – when a transaction starts. On the other hand, two-phase locking decides the serialization order dynamically – according to the order in which objects are accessed. Timestamp ordering, and in particular multiversion timestamp ordering, is better than strict two-phase locking for read-only transactions. Two-phase locking is better when the operations in transactions are predominantly updates.

Some work uses the observation that timestamp ordering is beneficial for transactions with predominantly *read* operations and that locking is beneficial for transactions with more *writes* than *reads* as an argument for allowing hybrid schemes in which some transactions use timestamp ordering and others use locking for concurrency control. Readers who are interested in the use of mixed methods should read Bernstein *et al.* [1987].

The pessimistic methods differ in the strategy used when a conflicting access to an object is detected. Timestamp ordering aborts the transaction immediately, whereas locking makes the transaction wait – but with a possible later penalty of aborting to avoid deadlock.

When optimistic concurrency control is used, all transactions are allowed to proceed, but some are aborted when they attempt to commit, or in forward validation transactions are aborted earlier. This results in relatively efficient operation when there are few conflicts, but a substantial amount of work may have to be repeated when a transaction is aborted.

Locking has been in use for many years in database systems, but timestamp ordering has been used in the SDD-1 database system. Both methods have been used in file servers. However, historically, the predominant method of concurrency control of access to data in distributed systems is by locking – for example, as mentioned earlier, the CORBA Concurrency Control Service is based entirely on the use of locks. In particular, it provides hierarchic locking, which allows for mixed-granularity locking on hierarchically structured data.

Several research distributed systems, for example Argus [Liskov 1988] and Arjuna [Shrivastava *et al.* 1991], have explored the use of semantic locks, timestamp ordering and new approaches to long transactions.

Ellis *et al.* [1991] wrote a review of requirements for multi-user applications in which all users expect to see common views of objects being updated by any of the users. Many of the schemes provided notification of changes made by other users, but this is contrary to the idea of isolation.

Barghouti and Kaiser [1991] wrote a review of what are sometimes described as ‘advanced database applications’ – for example, cooperative CAD/CAM and software development systems. In such applications, transactions last for a long time, and users work on independent versions of objects that are checked out from a common database and checked in when the work is finished. The merging of versions requires cooperation between users.

Similarly, the above concurrency control mechanisms are not always adequate for twenty-first-century applications that enable users to share documents over the Internet. Many of the latter use optimistic forms of concurrency control followed by conflict resolution instead of aborting one of any pair of conflicting operations.

The following are some examples.

**Dropbox** • Dropbox [www.dropbox.com] is a cloud service that provides file backup and enables users to share files and folders, accessing them from anywhere. Dropbox uses an optimistic form of concurrency control, keeping track of consistency and preventing clashes between users’ updates – which are at the granularity of whole files. Thus if two users make concurrent updates to the same file, the first write will be accepted and the second rejected. However, Dropbox provides a version history to enable users to merge their updates manually or restore previous versions.

**Google apps** • Google Apps [www.google.com I] are listed in Figure 21.2. They include Google Docs, a cloud service that provides web-based applications (word processor, spreadsheet and presentation) that allow users to collaborate with one another by means of shared documents. If several people edit the same document simultaneously, they will see each other’s changes. In the case of a word processor document, users can see one another’s cursors and updates are shown at the level of individual characters as they are typed by any participant. Users are left to resolve any conflicts that occur, but conflicts are generally avoided because users are continuously aware of each other’s activities. In the case of a spreadsheet document, users’ cursors and changes are displayed and updated at the granularity of single cells. If two users access the same cell simultaneously, the last update wins.

**Wikipedia** • Concurrency control for editing is optimistic, allowing editors concurrent access to web pages in which the first write is accepted and a user making a subsequent write is shown an ‘edit conflict’ screen and asked to resolve the conflicts.

**Dynamo** • Amazon.com’s key-value storage service uses optimistic concurrency control with conflict resolution (see the box on the next page).

## Dynamo

Dynamo [DeCandia et al. 2007] is one of the storage services used by Amazon.com, whose platform serves tens of millions of customers at peak times, using tens of thousands of servers. Such a setting makes very strong demands in terms of performance, reliability and scalability. Dynamo was designed to support applications such as the shopping carts and best-seller lists that require only primary key access to a value in a data store. Data is heavily replicated with a view to providing the scalability and availability that are key to these services.

Dynamo uses single *get* and *put* operations rather than transactions and does not provide the isolation guarantee specified in the ACID properties. So as to improve availability, it also provides a weaker form of consistency – which is acceptable in the applications it supports.

Optimistic methods are used for concurrency control. In cases where versions differ, they must be reconciled. Application logic can be used to merge versions in the case of the shopping cart application.

Where application logic cannot be used, timestamp-based reconciliation is applied. Dynamo uses the rule that ‘last write wins’ – the version with the largest timestamp becomes the new one.

## 16.8 Summary

Transactions provide a means by which clients can specify sequences of operations that are atomic in the presence of other concurrent transactions and server crashes. The first aspect of atomicity is achieved by running transactions so that their effects are serially equivalent. The effects of committed transactions are recorded in permanent storage so that the transaction service can recover from process crashes. To allow transactions the ability to abort, without having harmful side effects on other transactions, executions must be strict – that is, reads and writes of one transaction must be delayed until other transactions that wrote the same objects have either committed or aborted. To allow transactions the choice of either committing or aborting, their operations are performed in tentative versions that cannot be accessed by other transactions. The tentative versions of objects are copied to the real objects and to permanent storage when a transaction commits.

Nested transactions are formed by structuring transactions from other subtransactions. Nesting is particularly useful in distributed systems because it allows concurrent execution of subtransactions in separate servers. Nesting also has the advantage of allowing independent recovery of parts of a transaction.

Operation conflicts form a basis for the derivation of concurrency control protocols. Protocols must not only ensure serializability but also allow for recovery by using strict executions to avoid problems associated with transactions aborting, such as cascading aborts.

Three alternative strategies are possible in scheduling an operation in a transaction. They are (1) to execute it immediately, (2) to delay it or (3) to abort it.

Strict two-phase locking uses the first two strategies, resorting to abortion only in the case of deadlock. It ensures serializability by ordering transactions according to when they access common objects. Its main drawback is that deadlocks can occur.

Timestamp ordering uses all three strategies to ensure serializability by ordering transactions' accesses to objects according to the time transactions start. This method cannot suffer from deadlocks and is advantageous for read-only transactions. However, transactions must be aborted when they arrive too late. Multiversion timestamp ordering is particularly effective.

Optimistic concurrency control allows transactions to proceed without any form of checking until they are completed. Transactions are validated before being allowed to commit. Backward validation requires the maintenance of multiple write sets of committed transactions, whereas forward validation must validate against active transactions and has the advantage that it allows alternative strategies for resolving conflicts. Starvation can occur due to repeated aborting of a transaction that fails validation in optimistic concurrency control and even in timestamp ordering.

## EXERCISES

- 16.1 The TaskBag is a service whose functionality is to provide a repository for 'task descriptions'. It enables clients running in several computers to carry out parts of a computation in parallel. A *master* process places descriptions of subtasks of a computation in the TaskBag, and *worker* processes select tasks from the TaskBag and carry them out, returning descriptions of their results to the TaskBag. The *master* then collects the results and combines them to produce the final result.

The TaskBag service provides the following operations:

|                 |  |
|-----------------|--|
| <i>setTask</i>  | allows clients to add task descriptions to the bag;      |
| <i>takeTask</i> | allows clients to take task descriptions out of the bag. |

A client makes the request *takeTask*, when a task is not available but may be available soon. Discuss the advantages and drawbacks of the following alternatives:

- i) The server can reply immediately, telling the client to try again later.
- ii) The server operation (and therefore the client) must wait until a task becomes available.
- iii) Callbacks are used.

page 678

- 16.2 A server manages the objects  $a_1, a_2, \dots, a_n$ . The server provides two operations for its clients:

|                                |                              |
|--------------------------------|------------------------------|
| <i>read</i> ( $i$ )            | returns the value of $a_i$ ; |
| <i>write</i> ( $i$ , $Value$ ) | assigns $Value$ to $a_i$ .   |

The transactions  $T$  and  $U$  are defined as follows:

$T: x = \text{read}(j); y = \text{read}(i); \text{write}(j, 44); \text{write}(i, 33);$   
 $U: x = \text{read}(k); \text{write}(i, 55); y = \text{read}(j); \text{write}(k, 66).$

Give three serially equivalent interleavings of the transactions  $T$  and  $U$ . page 685

- 16.3 Give serially equivalent interleavings of  $T$  and  $U$  in Exercise 16.2 with the following properties:
- i) that are strict;
  - ii) that are not strict but could not produce cascading aborts;
  - iii) that could produce cascading aborts. page 689
- 16.4 The operation *create* inserts a new bank account at a branch. The transactions  $T$  and  $U$  are defined as follows:
- $T: aBranch.create("Z");$   
 $U: z.deposit(10); z.deposit(20).$

Assume that  $Z$  does not yet exist. Assume also that the *deposit* operation does nothing if the account given as the argument does not exist. Consider the following interleaving of transactions  $T$  and  $U$ :

| $T$                  | $U$              |
|----------------------|------------------|
|                      | $z.deposit(10);$ |
| $aBranch.create(Z);$ |                  |
|                      | $z.deposit(20);$ |

State the balance of  $Z$  after their execution in this order. Are these consistent with serially equivalent executions of  $T$  and  $U$ ? page 685

- 16.5 A newly created object like  $Z$  in Exercise 16.4 is sometimes called a *phantom*. From the point of view of transaction  $U$ ,  $Z$  is not there at first and then appears (like a ghost). Explain, with an example, how a phantom could occur when an account is deleted.
- 16.6 The ‘transfer’ transactions  $T$  and  $U$  are defined as:
- $T: a.withdraw(4); b.deposit(4);$   
 $U: c.withdraw(3); b.deposit(3);$
- Suppose that they are structured as pairs of nested transactions:
- $T_1: a.withdraw(4); T_2: b.deposit(4);$   
 $U_1: c.withdraw(3); U_2: b.deposit(3);$
- Compare the number of serially equivalent interleavings of  $T_1, T_2, U_1$  and  $U_2$  with the number of serially equivalent interleavings of  $T$  and  $U$ . Explain why the use of these nested transactions generally permits a larger number of serially equivalent interleavings than non-nested ones. page 685
- 16.7 Consider the recovery aspects of the nested transactions defined in Exercise 16.6. Assume that a *withdraw* transaction will abort if the account will be overdrawn and that in this case the parent transaction will also abort. Describe serially equivalent interleavings of  $T_1, T_2, U_1$  and  $U_2$  with the following properties:
- i) that are strict;
  - ii) that are not strict.
- To what extent does the criterion of strictness reduce the potential concurrency gain of nested transactions? page 685



- 16.8 Explain why serial equivalence requires that once a transaction has released a lock on an object, it is not allowed to obtain any more locks.

A server manages the objects  $a_1, a_2, \dots, a_n$ . The server provides two operations for its clients:

$read(i)$  returns the value of  $a_i$

$write(i, Value)$  assigns  $Value$  to  $a_i$

The transactions  $T$  and  $U$  are defined as follows:

$T: x = read(i); write(j, 44);$

$U: write(i, 55); write(j, 66);$

Describe an interleaving of the transactions  $T$  and  $U$  in which locks are released early with the effect that the interleaving is not serially equivalent. page 693

- 16.9 The transactions  $T$  and  $U$  at the server in Exercise 16.8 are defined as follows:

$T: x = read(i); write(j, 44);$

$U: write(i, 55); write(j, 66);$

Initial values of  $a_i$  and  $a_j$  are 10 and 20, respectively. Which of the following interleavings are serially equivalent, and which could occur with two-phase locking?

| (a) <table border="1"> <thead> <tr> <th><math>T</math></th> <th><math>U</math></th> </tr> </thead> <tbody> <tr> <td><math>x = read(i);</math></td> <td></td> </tr> <tr> <td></td> <td><math>write(i, 55);</math></td> </tr> <tr> <td><math>write(j, 44);</math></td> <td></td> </tr> <tr> <td></td> <td><math>write(j, 66);</math></td> </tr> </tbody> </table> | $T$             | $U$ | $x = read(i);$ |                 |  | $write(i, 55);$ | $write(j, 44);$ |  |                 | $write(j, 66);$ | (b) <table border="1"> <thead> <tr> <th><math>T</math></th> <th><math>U</math></th> </tr> </thead> <tbody> <tr> <td><math>x = read(i);</math></td> <td></td> </tr> <tr> <td><math>write(j, 44);</math></td> <td></td> </tr> <tr> <td></td> <td><math>write(i, 55);</math></td> </tr> <tr> <td></td> <td><math>write(j, 66);</math></td> </tr> </tbody> </table> | $T$ | $U$ | $x = read(i);$ |                 | $write(j, 44);$ |  |  | $write(i, 55);$ |                 | $write(j, 66);$ |
|---|-----------------|-----|----------------|-----------------|--|-----------------|-----------------|--|-----------------|-----------------|---|-----|-----|----------------|-----------------|-----------------|--|--|-----------------|-----------------|-----------------|
| $T$   | $U$             |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| $x = read(i);$  |                 |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
|   | $write(i, 55);$ |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| $write(j, 44);$   |                 |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
|   | $write(j, 66);$ |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| $T$   | $U$             |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| $x = read(i);$  |                 |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| $write(j, 44);$   |                 |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
|   | $write(i, 55);$ |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
|   | $write(j, 66);$ |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| (c) <table border="1"> <thead> <tr> <th><math>T</math></th> <th><math>U</math></th> </tr> </thead> <tbody> <tr> <td></td> <td><math>write(i, 55);</math></td> </tr> <tr> <td></td> <td><math>write(j, 66);</math></td> </tr> <tr> <td><math>x = read(i);</math></td> <td></td> </tr> <tr> <td><math>write(j, 44);</math></td> <td></td> </tr> </tbody> </table> | $T$             | $U$ |                | $write(i, 55);$ |  | $write(j, 66);$ | $x = read(i);$  |  | $write(j, 44);$ |                 | (d) <table border="1"> <thead> <tr> <th><math>T</math></th> <th><math>U</math></th> </tr> </thead> <tbody> <tr> <td></td> <td><math>write(i, 55);</math></td> </tr> <tr> <td><math>x = read(i);</math></td> <td></td> </tr> <tr> <td></td> <td><math>write(j, 66);</math></td> </tr> <tr> <td><math>write(j, 44);</math></td> <td></td> </tr> </tbody> </table> | $T$ | $U$ |                | $write(i, 55);$ | $x = read(i);$  |  |  | $write(j, 66);$ | $write(j, 44);$ |                 |
| $T$   | $U$             |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
|   | $write(i, 55);$ |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
|   | $write(j, 66);$ |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| $x = read(i);$  |                 |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| $write(j, 44);$   |                 |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| $T$   | $U$             |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
|   | $write(i, 55);$ |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| $x = read(i);$  |                 |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
|   | $write(j, 66);$ |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |
| $write(j, 44);$   |                 |     |                |                 |  |                 |                 |  |                 |                 |   |     |     |                |                 |                 |  |  |                 |                 |                 |

page 693

- 16.10 Consider a relaxation of two-phase locks in which read-only transactions can release read locks early. Would a read-only transaction have consistent retrievals? Would the objects become inconsistent? Illustrate your answer with the following transactions  $T$  and  $U$  at the server in Exercise 16.8:

$T: x = read(i); y = read(j);$

$U: write(i, 55); write(j, 66);$

in which initial values of  $a_i$  and  $a_j$  are 10 and 20.

page 690

- 16.11 The executions of transactions are strict if *read* and *write* operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted. Explain how the locking rules in Figure 16.16 ensure strict executions. page 696
- 16.12 Describe how a non-recoverable situation could arise if write locks are released after the last operation of a transaction but before its commitment. page 690
- 16.13 Explain why executions are always strict, even if read locks are released after the last operation of a transaction but before its commitment. Give an improved statement of rule 2 in Figure 16.16. page 690
- 16.14 Consider a deadlock detection scheme for a single server. Describe precisely when edges are added to and removed from the wait-for-graph.  
Illustrate your answer with respect to the following transactions *T*, *U* and *V* at the server of Exercise 16.8.

| <i>T</i>                      | <i>U</i>                      | <i>V</i>                      |
|-------------------------------|-------------------------------|-------------------------------|
|                               | <i>write</i> ( <i>i</i> , 66) |                               |
| <i>write</i> ( <i>i</i> , 55) |                               |                               |
|                               |                               | <i>write</i> ( <i>i</i> , 77) |
|                               | <i>commit</i>                 |                               |

- When *U* releases its write lock on *a<sub>i</sub>*, both *T* and *V* are waiting to obtain write locks on it. Does your scheme work correctly if *T* (first come) is granted the lock before *V*? If your answer is ‘No’, then modify your description. page 702
- 16.15 Consider hierarchic locks as illustrated in Figure 16.26. What locks must be set when an appointment is assigned to a time slot in week *w*, day *d*, at time *t*? In what order should these locks be set? Does the order in which they are released matter?  
What locks must be set when the time slots for every day in week *w* are viewed? Can this be done when the locks for assigning an appointment to a time slot are already set? page 705
- 16.16 Consider optimistic concurrency control as applied to the transactions *T* and *U* defined in Exercise 16.9. Suppose that transactions *T* and *U* are active at the same time as one another. Describe the outcome in each of the following cases:
- i) *T*’s request to commit comes first and backward validation is used.
  - ii) *U*’s request to commit comes first and backward validation is used.
  - iii) *T*’s request to commit comes first and forward validation is used.
  - iv) *U*’s request to commit comes first and forward validation is used.
- In each case describe the sequence in which the operations of *T* and *U* are performed, remembering that writes are not carried out until after validation.

16.17 Consider the following interleaving of transactions  $T$  and  $U$ :

| $T$                    | $U$                    |
|------------------------|------------------------|
| <i>openTransaction</i> | <i>openTransaction</i> |
| $y = \text{read}(k);$  |                        |
|                        | $\text{write}(i, 55);$ |
|                        | $\text{write}(j, 66);$ |
|                        | <i>commit</i>          |
| $x = \text{read}(i);$  |                        |
| $\text{write}(j, 44);$ |                        |

The outcome of optimistic concurrency control with backward validation is that  $T$  will be aborted because its *read* operation conflicts with  $U$ 's *write* operation on  $a_i$ , although the interleavings are serially equivalent. Suggest a modification to the algorithm that deals with such cases. page 707

16.18 Make a comparison of the sequences of operations of the transactions  $T$  and  $U$  of Exercise 16.8 that are possible under two-phase locking (Exercise 16.9) and under optimistic concurrency control (Exercise 16.16). page 707

16.19 Consider the use of timestamp ordering with each of the example interleavings of transactions  $T$  and  $U$  in Exercise 16.9. Initial values of  $a_i$  and  $a_j$  are 10 and 20, respectively, and initial read and write timestamps are  $t_0$ . Assume that each transaction opens and obtains a timestamp just before its first operation; for example, in (a)  $T$  and  $U$  get timestamps  $t_1$  and  $t_2$ , respectively, where  $t_0 < t_1 < t_2$ . Describe in order of increasing time the effects of each operation of  $T$  and  $U$ . For each operation, state the following:

- whether the operation may proceed according to the write or read rule;
- when timestamps are assigned to transactions or objects;
- when tentative objects are created and when their values are set.

What are the final values of the objects and their timestamps?

page 711

16.20 Repeat Exercise 16.19 for the following interleavings of transactions  $T$  and  $U$ :

| $T$                    | $U$                    |
|------------------------|------------------------|
| <i>openTransaction</i> |                        |
|                        | <i>openTransaction</i> |
|                        | $\text{write}(i, 55);$ |
|                        | $\text{write}(j, 66);$ |
| $x = \text{read}(i);$  |                        |
| $\text{write}(j, 44);$ |                        |
|                        | <i>commit</i>          |

| $T$                    | $U$                    |
|------------------------|------------------------|
| <i>openTransaction</i> |                        |
|                        | <i>openTransaction</i> |
|                        | $\text{write}(i, 55);$ |
|                        | $\text{write}(j, 66);$ |
|                        | <i>commit</i>          |
| $x = \text{read}(i);$  |                        |
| $\text{write}(j, 44);$ |                        |

page 711

- 16.21 Repeat Exercise 16.20 using multiversion timestamp ordering. *page 715*
- 16.22 In multiversion timestamp ordering, *read* operations can access tentative versions of objects. Give an example to show how cascading aborts can happen if all *read* operations are allowed to proceed immediately. *page 715*
- 16.23 What are the advantages and drawbacks of multiversion timestamp ordering in comparison with ordinary timestamp ordering? *page 715*

## DISTRIBUTED TRANSACTIONS

- 17.1 Introduction
- 17.2 Flat and nested distributed transactions
- 17.3 Atomic commit protocols
- 17.4 Concurrency control in distributed transactions
- 17.5 Distributed deadlocks
- 17.6 Transaction recovery
- 17.7 Summary

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

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

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

The use of locking schemes can lead to distributed deadlocks. Distributed deadlock detection algorithms are discussed in Section 17.5.

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

## 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

---

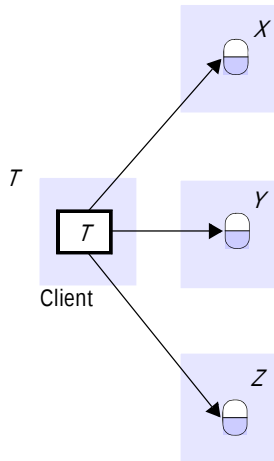
A client transaction becomes distributed if it invokes operations in several different servers. There are two different ways that distributed transactions can be structured: as flat transactions and as nested transactions.

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.

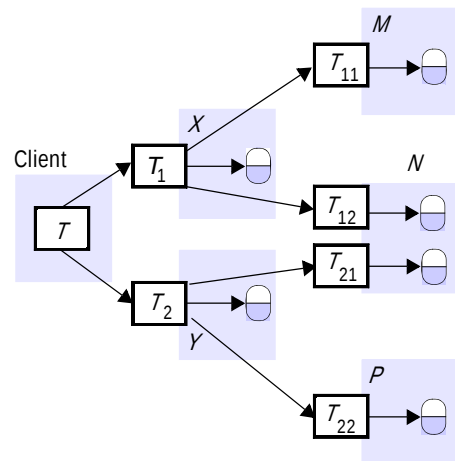
In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting. Figure 17.1(b) shows a client transaction  $T$  that opens two subtransactions,  $T_1$  and  $T_2$ , which access objects at servers  $X$  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

(a) Flat transaction



(b) Nested 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 *withdrawals*) 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

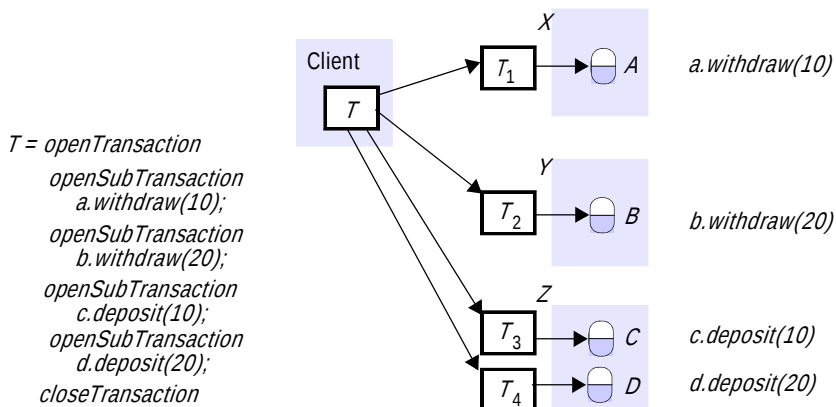
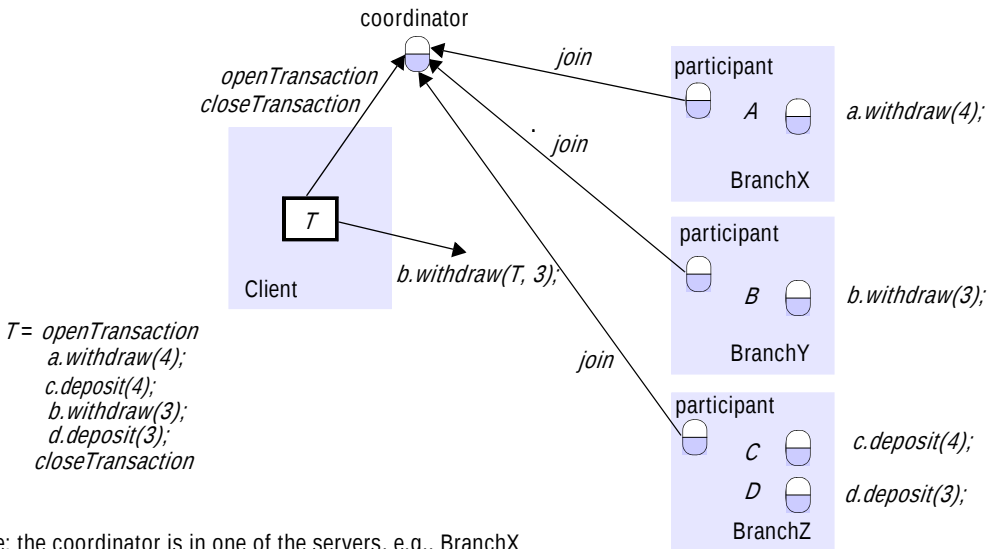


Figure 17.3 A distributed banking transaction



### 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.

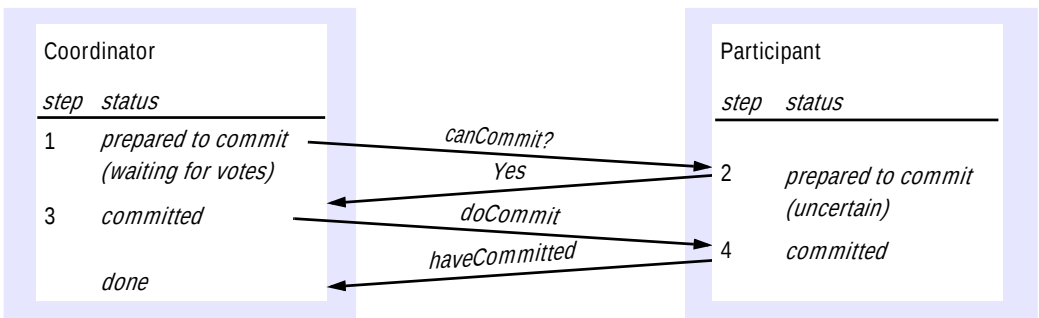
*Phase 2 (completion according to outcome of vote):*

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*<sub>1</sub>, *T*<sub>2</sub>, *T*<sub>11</sub>, *T*<sub>12</sub>, *T*<sub>21</sub> and *T*<sub>22</sub> are subtransactions. *T*<sub>1</sub> and *T*<sub>2</sub> are child transactions of *T*, which is referred to as their *parent*. Similarly, *T*<sub>11</sub> and *T*<sub>12</sub> are child transactions of *T*<sub>1</sub>, and *T*<sub>21</sub> and *T*<sub>22</sub> are child transactions of *T*<sub>2</sub>. Each subtransaction starts after its parent and finishes before it. Thus, for example, *T*<sub>11</sub> and *T*<sub>12</sub> start after *T*<sub>1</sub> and finish before it.

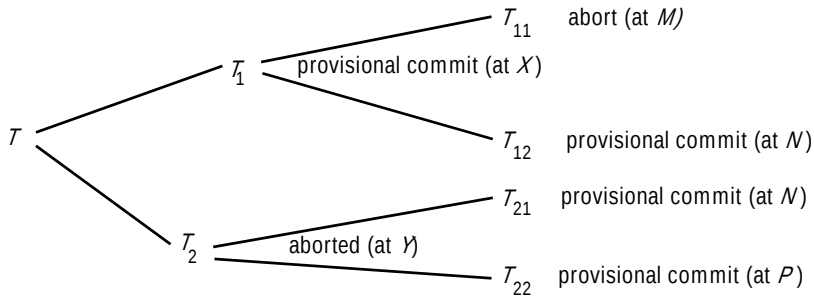
When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. A provisional commit is different from being prepared to commit: nothing is backed up in permanent storage. If the server crashes subsequently, its replacement will not be able to commit. After 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

| <i>Coordinator of transaction</i> | <i>Child transactions</i> | <i>Participant</i>          | <i>Provisional commit list</i> | <i>Abort list</i> |
|-----------------------------------|---------------------------|-----------------------------|--------------------------------|-------------------|
| $T$                               | $T_1, T_2$                | 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 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  $\langle local\ timestamp, server-id \rangle$  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:

| $T$                  |        | $U$                  |        |
|----------------------|--------|----------------------|--------|
| <i>read</i> ( $A$ )  | at $X$ | <i>read</i> ( $B$ )  | at $Y$ |
| <i>write</i> ( $A$ ) |        | <i>write</i> ( $B$ ) |        |
| <i>read</i> ( $B$ )  | at $Y$ | <i>read</i> ( $A$ )  | at $X$ |
| <i>write</i> ( $B$ ) |        | <i>write</i> ( $A$ ) |        |

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*

| <i>U</i>               |                  | <i>V</i>               |                  | <i>W</i>               |                  |
|------------------------|------------------|------------------------|------------------|------------------------|------------------|
| <i>d.deposit</i> (10)  | lock <i>D</i>    |                        |                  |                        |                  |
|                        |                  | <i>b.deposit</i> (10)  | lock <i>B</i>    |                        |                  |
| <i>a.deposit</i> (20)  | lock <i>A</i>    |                        | at <i>Y</i>      |                        |                  |
|                        | at <i>X</i>      |                        |                  |                        |                  |
|                        |                  |                        |                  | <i>c.deposit</i> (30)  | lock <i>C</i>    |
| <i>b.withdraw</i> (30) | wait at <i>Y</i> |                        |                  |                        | at <i>Z</i>      |
|                        |                  | <i>c.withdraw</i> (20) | wait at <i>Z</i> |                        |                  |
|                        |                  |                        |                  | <i>a.withdraw</i> (20) | wait at <i>X</i> |

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:

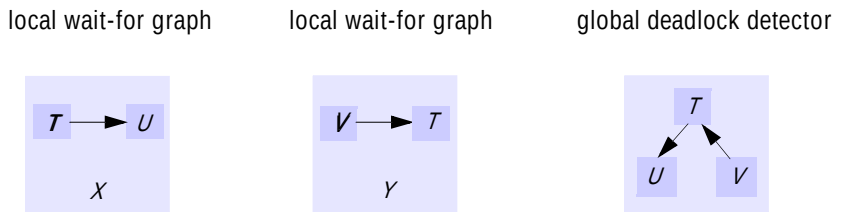
- server *Y*: *U* → *V* (added when *U* requests *b.withdraw*(30))
- server *Z*: *V* → *W* (added when *V* requests *c.withdraw*(20))
- server *X*: *W* → *U* (added when *W* requests *a.withdraw*(20))

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 17.14 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.

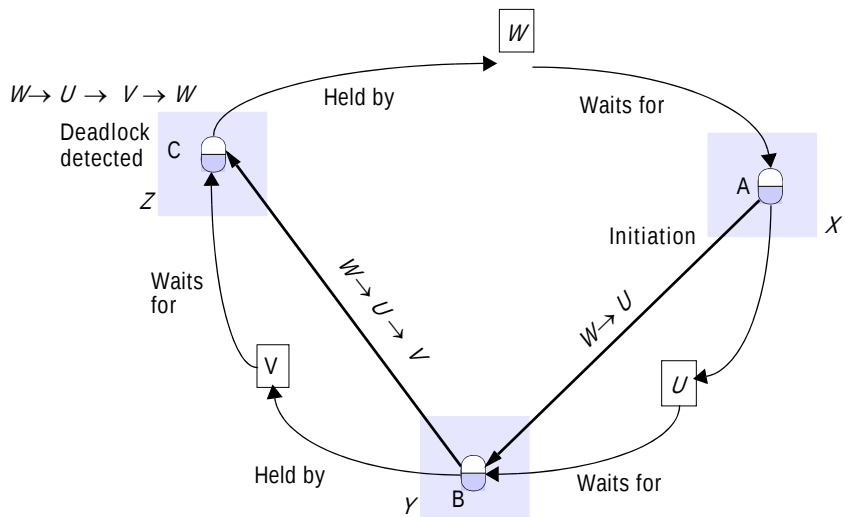
The question is, when should a server send out a probe? Consider the situation at server  $X$  in Figure 17.13. This server has just added the edge  $W \rightarrow U$  to its local wait-for graph, and at this time transaction  $U$  is waiting to access object  $B$ , which transaction  $V$  holds at server  $Y$ . This edge could possibly be part of a cycle such as  $V \rightarrow T_1 \rightarrow T_2 \rightarrow \dots \rightarrow W \rightarrow U \rightarrow V$  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  $\langle T \rightarrow U \rangle$  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  $\langle T \rightarrow U \rangle$  (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  $\langle T \rightarrow U \rightarrow V \rangle$ ), 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,  $\langle T \rightarrow U \rightarrow V \rightarrow T \rangle$ ). 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  $\langle W \rightarrow U \rangle$  to the server of *B* (Server *Y*).
- Server *Y* receives probe  $\langle W \rightarrow U \rangle$ , notes that *B* is held by *V* and appends *V* to the probe to produce  $\langle W \rightarrow U \rightarrow V \rangle$ . It notes that *V* is waiting for *C* at server *Z*. This probe is forwarded to server *Z*.
- Server *Z* receives probe  $\langle W \rightarrow U \rightarrow V \rangle$ , notes *C* is held by *W* and appends *W* to the probe to produce  $\langle W \rightarrow U \rightarrow V \rightarrow W \rangle$ .

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.

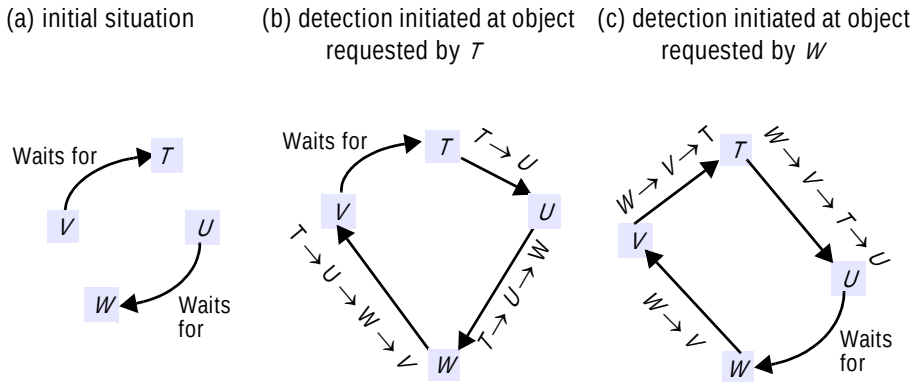
Figure 17.15 shows the progress of the probe messages from the initiation by the server of *A* to the deadlock detection by the server of *C*. Probes are shown as heavy arrows, objects as ovals (as usual) and transaction coordinators as rectangles. Each probe is shown as going directly from one object 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  $\langle W \rightarrow U \rightarrow V \rangle$  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  $\langle W \rightarrow U \rightarrow V \rangle$  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  $T$  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,  $\langle T \rightarrow U \rangle$  and  $\langle W \rightarrow V \rangle$ , 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  $\langle T \rightarrow U \rightarrow W \rightarrow V \rightarrow T \rangle$ , and in Figure 17.16 (c), the cycle is  $\langle W \rightarrow V \rightarrow T \rightarrow U \rightarrow W \rangle$ .

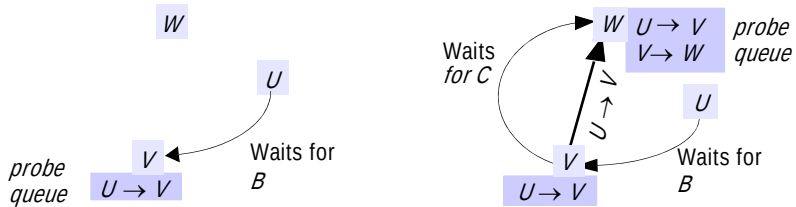
In order to ensure that only one transaction in a cycle is aborted, transactions are given *priorities* in such a way that all transactions are totally ordered. Timestamps, for example, may be used as priorities. When a deadlock cycle is found, the transaction with the lowest priority is aborted. Even 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  $\langle T \rightarrow U \rightarrow W \rightarrow V \rightarrow T \rangle$  or  $\langle W \rightarrow V \rightarrow T \rightarrow U \rightarrow W \rangle$  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  $\langle T \rightarrow U \rangle$  would be sent, but as  $W < V$  the initiating probe  $\langle W \rightarrow V \rangle$  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

(a)  $V$  stores probe when  $U$  starts waiting (b) Probe is forwarded when  $V$  starts waiting



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  $\langle W \rightarrow U \rangle$ . 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  $\langle U \rightarrow V \rangle$  – see Figure 17.17(a). Then when  $V$  starts waiting for  $W$ , the coordinator of  $W$  will store  $\langle V \rightarrow W \rangle$  and  $V$  will forward its probe queue,  $\langle U \rightarrow V \rangle$ , to  $W$ . (See Figure 17.17(b), in which  $W$ 's probe queue has  $\langle U \rightarrow V \rangle$  and  $\langle V \rightarrow W \rangle$ .) When  $W$  starts waiting for  $A$  it will forward its probe queue,  $\langle U \rightarrow V \rightarrow W \rangle$ , 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 ( <i>prepared</i> , <i>committed</i> , <i>aborted</i> ) 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 $\langle \text{objectID}, P_i \rangle$ , 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.

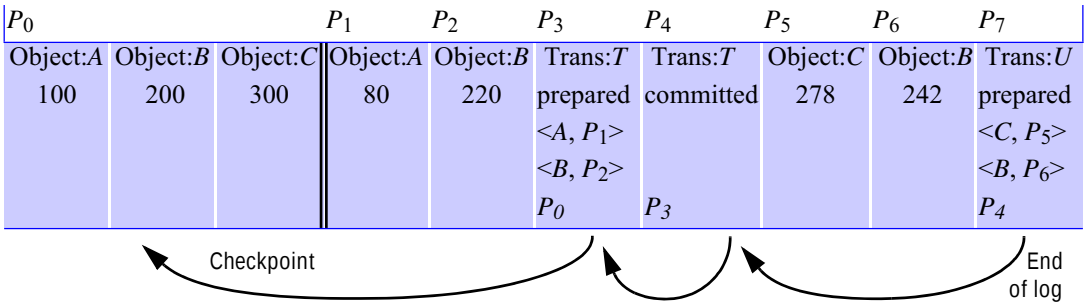
It is assumed that the append operation is atomic in the sense that it writes one or more complete entries to the recovery file. If the server fails, only the last write can be incomplete. To make efficient use of the disk, several subsequent writes can be buffered and then written to disk as a single 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 ( $\langle A, P_1 \rangle, \langle B, P_2 \rangle$ ). 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 ( $\langle C, P_5 \rangle, \langle B, P_6 \rangle$ ).

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*<sub>1</sub> and *P*<sub>2</sub> made by transaction *T*. It also contains the shadow versions of *B* and *C* made by transaction *U*, at *P*<sub>3</sub> and *P*<sub>4</sub>.

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                 | Transaction status file (stable storage) |           |                     |
|---------------------|--|-----------|---------------------|
|                     | <i>T</i>                                 | <i>T</i>  | <i>U</i>            |
| $A \rightarrow P_1$ | prepared                                 | committed | prepared            |
| $B \rightarrow P_2$ | $A \rightarrow P_1$                      |           | $B \rightarrow P_3$ |
| $C \rightarrow P_0$ | $B \rightarrow P_2$                      |           | $C \rightarrow P_4$ |

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             |
|--------------------|--|
| <i>Coordinator</i> | Transaction identifier, list of participants |
| <i>Participant</i> | 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

|   |   |   |                              |   |   |   |                                       |                              |                              |
|---|---|---|------------------------------|---|---|---|---------------------------------------|------------------------------|------------------------------|
| Trans: <i>T</i><br>prepared<br>intentions<br>list | Coord'r: <i>T</i><br>part'pant<br>list: . . . | • | Trans: <i>T</i><br>committed | Trans: <i>U</i><br>prepared<br>intentions<br>list | • | • | Part'pant: <i>U</i><br>Coord'r: . . . | Trans: <i>U</i><br>uncertain | Trans: <i>U</i><br>committed |
|---|---|---|------------------------------|---|---|---|---------------------------------------|------------------------------|------------------------------|

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.

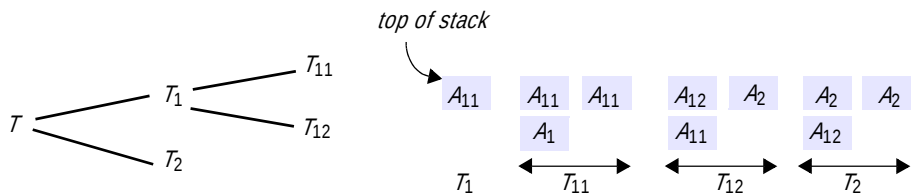
Figure 17.22 Recovery of the two-phase commit protocol

| Role        | Status           | Action of recovery manager   |
|-------------|------------------|--|
| Coordinator | <i>prepared</i>  | No decision had been reached before the server failed. It sends <i>abortTransaction</i> to all the servers in the participant list and adds the transaction status <i>aborted</i> in its recovery file. Same action for state <i>aborted</i> . If there is no participant list, the participants will eventually time out and abort the transaction. |
| Coordinator | <i>committed</i> | A decision to commit had been reached before the server failed. It sends a <i>doCommit</i> 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 | <i>committed</i> | The participant sends a <i>haveCommitted</i> 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 | <i>uncertain</i> | 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 <i>getDecision</i> to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.               |
| Participant | <i>prepared</i>  | The participant has not yet voted and can abort the transaction.   |
| Coordinator | <i>done</i>      | 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.

## 17.7 Summary

In the most general case, a client's transaction will request operations on objects in several different servers. A distributed transaction is any transaction whose activity involves several different servers. A nested transaction structure may be used to allow additional concurrency and independent committing by the servers in a distributed transaction.

The atomicity property of transactions requires that the servers participating in a distributed transaction either all commit it or all abort it. Atomic commit protocols are designed to achieve this effect, even if servers crash during their execution. The two-phase commit protocol allows a server to decide to abort unilaterally. It includes timeout actions to deal with delays due to servers crashing. The two-phase commit protocol can take an unbounded amount of time to complete but is guaranteed to complete eventually.

Concurrency control in distributed transactions is modular – each server is responsible for the serializability of transactions that access its own objects. However, additional protocols are required to ensure that transactions are serializable globally. Distributed transactions that use timestamp ordering require a means of generating an agreed timestamp ordering between the multiple servers. Those that use optimistic concurrency control require global validation or a means of forcing a global ordering on committing transactions.

Distributed transactions that use two-phase locking can suffer from distributed deadlocks. The aim of distributed deadlock detection is to look for cycles in the global wait-for graph. If a cycle is found, one or more transactions must be aborted to resolve the deadlock. Edge chasing is a non-centralized approach to the detection of distributed deadlocks.

Transaction-based applications have strong requirements for the long life and integrity of the information stored, but they do not usually have requirements for immediate response at all times. Atomic commit protocols are the key to distributed transactions, but they cannot be guaranteed to complete within a particular time limit. Transactions are made durable by performing checkpoints and logging in a recovery file, which is used for recovery when a server is replaced after a crash. Users of a transaction service will experience some delay during recovery. Although it is assumed that the servers of distributed transactions exhibit crash failures and run in an asynchronous system, they are able to reach consensus about the outcome of transactions because crashed servers are replaced with new processes that can acquire all the relevant information from permanent storage or from other servers.

## EXERCISES

- 17.1 In a decentralized variant of the two-phase commit protocol the participants communicate directly with one another instead of indirectly via a coordinator. In phase 1, the coordinator sends its vote to all the participants. In phase 2, if the coordinator's vote is *No*, the participants just abort the transaction; if it is *Yes*, each participant sends its vote to the coordinator and the other participants, each of which decides on the outcome according to the vote and carries it out. Calculate the number of messages and the number of rounds it takes. What are its advantages and disadvantages in comparison with the centralized variant? page 732
- 17.2 A three-phase commit protocol has the following parts:
- Phase 1:* Is the same as for two-phase commit.
  - Phase 2:* The coordinator collects the votes and makes a decision. If it is *No*, it *aborts* and informs participants that voted *Yes*; if the decision is *Yes*, it sends a *preCommit* request to all the participants. Participants that voted *Yes* wait for a *preCommit* or *doAbort* request. They acknowledge *preCommit* requests and carry out *doAbort* requests.
  - Phase 3:* The coordinator collects the acknowledgements. When all are received, it *commits* and sends *doCommit* requests to the participants. Participants wait for a *doCommit* request. When it arrives, they *commit*.
- Explain how this protocol avoids delay to participants during their 'uncertain' period due to the failure of the coordinator or other participants. Assume that communication does not fail. page 735
- 17.3 Explain how the two-phase commit protocol for nested transactions ensures that if the top-level transaction commits, all the right descendants are committed or aborted. page 736
- 17.4 Give an example of the interleaving, of two transactions that is serially equivalent at each server but is not serially equivalent globally. page 740



- 17.5 The *getDecision* procedure defined in Figure 17.4 is provided only by coordinators. Define a new version of *getDecision* to be provided by participants for use by other participants that need to obtain a decision when the coordinator is unavailable.

Assume that any active participant can make a *getDecision* request to any other active participant. Does this solve the problem of delay during the ‘uncertain’ period? Explain your answer. At what point in the two-phase commit protocol would the coordinator inform the participants of the other participants’ identities (to enable this communication)? page 732

- 17.6 Extend the definition of two-phase locking to apply to distributed transactions. Explain how this is ensured by distributed transactions using strict two-phase locking locally. page 740, Chapter 16

- 17.7 Assuming that strict two-phase locking is in use, describe how the actions of the two-phase commit protocol relate to the concurrency control actions of each individual server. How does distributed deadlock detection fit in? pages 732, 740

- 17.8 A server uses timestamp ordering for local concurrency control. What changes must be made to adapt it for use with distributed transactions? Under what conditions could it be argued that the two-phase commit protocol is redundant with timestamp ordering? pages 732, 741

- 17.9 Consider distributed optimistic concurrency control in which each server performs local backward validation sequentially (that is, with only one transaction in the validate and update phase at one time), in relation to your answer to Exercise 17.4. Describe the possible outcomes when the two transactions attempt to commit. What difference does it make if the servers use parallel validation? Chapter 16, page 742

- 17.10 A centralized global deadlock detector holds the union of local wait-for graphs. Give an example to explain how a phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure. page 745

- 17.11 Consider the edge-chasing algorithm (without priorities). Give examples to show that it could detect phantom deadlocks. page 746

- 17.12 A server manages the objects  $a_1, a_2, \dots, a_n$ . It provides two operations for its clients:

*read*( $i$ ) returns the value of  $a_i$   
*write*( $i, Value$ ) assigns *Value* to  $a_i$

The transactions  $T$ ,  $U$  and  $V$  are defined as follows:

$T: x = \text{read}(i); \text{write}(j, 44);$   
 $U: \text{write}(i, 55); \text{write}(j, 66);$   
 $V: \text{write}(k, 77); \text{write}(k, 88);$

Describe the information written to the log file on behalf of these three transactions if strict two-phase locking is in use and  $U$  acquires  $a_i$  and  $a_j$  before  $T$ . Describe how the recovery manager would use this information to recover the effects of  $T$ ,  $U$  and  $V$  when the server is replaced after a crash. What is the significance of the order of the commit entries in the log file? pages 753–754

- 17.13 The appending of an entry to the log file is atomic, but append operations from different transactions may be interleaved. How does this affect the answer to Exercise 17.12? pages 753–754

17.14 The transactions  $T$ ,  $U$  and  $V$  of Exercise 17.12 use strict two-phase locking and their requests are interleaved as follows:

| $T$                   | $U$                   | $V$                    |
|-----------------------|-----------------------|------------------------|
| $x = \text{read}(i);$ |                       |                        |
|                       |                       | $\text{write}(k, 77);$ |
|                       | $\text{write}(i, 55)$ |                        |
| $\text{write}(j, 44)$ |                       |                        |
|                       |                       | $\text{write}(k, 88)$  |
|                       | $\text{write}(j, 66)$ |                        |

- Assuming that the recovery manager appends the data entry corresponding to each *write* operation to the log file immediately instead of waiting until the end of the transaction, describe the information written to the log file on behalf of the transactions  $T$ ,  $U$  and  $V$ . Does early writing affect the correctness of the recovery procedure? What are the advantages and disadvantages of early writing? pages 753–754
- 17.15 Transactions  $T$  and  $U$  are run with timestamp ordering concurrency control. Describe the information written to the log file on behalf of  $T$  and  $U$ , allowing for the fact that  $U$  has a later timestamp than  $T$  and must wait to commit after  $T$ . Why is it essential that the commit entries in the log file be ordered by timestamps? Describe the effect of recovery if the server crashes (i) between the two *Commits*; (ii) after both of them.

| $T$                    | $U$                    |
|------------------------|------------------------|
| $x = \text{read}(i);$  |                        |
|                        | $\text{write}(i, 55);$ |
|                        | $\text{write}(j, 66);$ |
| $\text{write}(j, 44);$ |                        |
|                        | <i>Commit</i>          |
| <i>Commit</i>          |                        |

- What are the advantages and disadvantages of early writing with timestamp ordering? page 757
- 17.16 The transactions  $T$  and  $U$  in Exercise 17.15 are run with optimistic concurrency control using backward validation and restarting any transactions that fail. Describe the information written to the log file on their behalf. Why is it essential that the commit entries in the log file be ordered by transaction numbers? How are the write sets of committed transactions represented in the log file? pages 753–754
- 17.17 Suppose that the coordinator of a transaction crashes after it has recorded the intentions list entry but before it has recorded the participant list or sent out the *canCommit*? requests. Describe how the participants resolve the situation. What will the coordinator do when it recovers? Would it be any better to record the participant list before the intentions list entry? page 758