16

# TRANSACTIONS AND CONCURRENCY CONTROL

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 XDFS [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:

**A**tomicity: a transaction must be all or nothing;

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

**I**solation;

**D**urability.

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* | |
|---|---|---|---|
| *openTransaction* | *openTransaction* | | *openTransaction* |
| *operation* | *operation* | | *operation* |
| *operation* | *operation* | | *operation* |
| • | • | server aborts | • |
| • | • | *transaction* → | • |
| *operation* | *operation* | | *operation ERROR* |
| | | | *reported to client* |
| *closeTransaction* | *abortTransaction* | | |

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*:** | |
|---|---|---|---|
| *balance = b.getBalance();* | | *balance = b.getBalance();* | |
| *b.setBalance(balance*1.1);* | | *b.setBalance(balance*1.1);* | |
| *a.withdraw(balance/10)* | | *c.withdraw(balance/10)* | |
| *balance = b.getBalance();* | $200 | | |
| | | *balance = b.getBalance();* | $200 |
| | | *b.setBalance(balance*1.1);* | $220 |
| *b.setBalance(balance*1.1);* | $220 | | |
| *a.withdraw(balance/10)* | $80 | | |
| | | *c.withdraw(balance/10)* | $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*:** | |
|---|---|---|---|
| *a.withdraw(100)* | | *aBranch.branchTotal()* | |
| *b.deposit(100)* | | | |
| *a.withdraw(100);* | $100 | | |
| | | *total = a.getBalance()* | $100 |
| | | *total = total + b.getBalance()* | $300 |
| | | *total = total + c.getBalance()* | |
| *b.deposit(100)* | $300 | • | |
| | | • | |

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

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *balance = b.getBalance()* | | *balance = b.getBalance()* | |
| *b.setBalance(balance*1.1)* | | *b.setBalance(balance*1.1)* | |
| *a.withdraw(balance/10)* | | *c.withdraw(balance/10)* | |
| *balance = b.getBalance()* | $200 | | |
| *b.setBalance(balance*1.1)* | $220 | | |
| | | *balance = b.getBalance()* | $220 |
| | | *b.setBalance(balance*1.1)* | $242 |
| *a.withdraw(balance/10)* | $80 | | |
| | | *c.withdraw(balance/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*:** | |
|---|---|---|---|
| *a.withdraw(100);* *b.deposit(100)* | | *aBranch.branchTotal( )* | |
| *a.withdraw(100);* | $100 | | |
| *b.deposit(100)* | $300 | | |
| | | *total = a.getBalance( )* | $100 |
| | | *total = total + b.getBalance()* | $400 |
| | | *total = total + c.getBalance()* | |
| | | ... | |

**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* |
|---|---|---|---|
| *read* | *read* | No | Because the effect of a pair of *read* operations does not depend on the order in which they are executed |
| *read* | *write* | Yes | Because the effect of a *read* and a *write* operation depends on the order of their execution |
| *write* | *write* | Yes | Because the effect of a pair of *write* 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 = read(i)* | |
| *write(i, 10)* | |
| | *y = read(j)* |
| | *write(j, 30)* |
| *write(j, 20)* | |
| | *z = 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 *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *a.getBalance()* | | *a.getBalance()* | |
| *a.setBalance(balance + 10)* | | *a.setBalance(balance + 20)* | |
| *balance = a.getBalance()* | $100 | | |
| *a.setBalance(balance + 10)* | $110 | | |
| | | *balance = a.getBalance()* | $110 |
| | | *a.setBalance(balance + 20)* | $130 |
| | | *commit transaction* | |
| *abort transaction* | | | |

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 *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *a.setBalance(105)* | | *a.setBalance(110)* | |
| | $100 | | |
| *a.setBalance(105)* | $105 | | |
| | | *a.setBalance(110)* | $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.

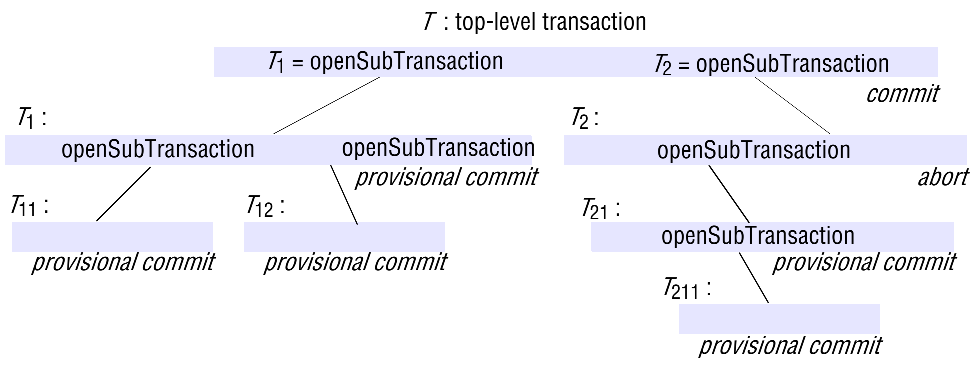
## 16.3 Nested transactions

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*:** | |
|---|---|---|---|
| *balance = b.getBalance()* | | *balance = b.getBalance()* | |
| *b.setBalance(bal*1.1)* | | *b.setBalance(bal*1.1)* | |
| *a.withdraw(bal/10)* | | *c.withdraw(bal/10)* | |
| Operations | Locks | Operations | Locks |
| *openTransaction* | | | |
| *bal = b.getBalance()* | lock *B* | | |
| *b.setBalance(bal*1.1)* | | *openTransaction* | |
| *a.withdraw(bal/10)* | lock *A* | *bal = b.getBalance()* | waits for *T*'s lock on *B* |
| *closeTransaction* | unlock *A*, *B* | • • • | |
| | | | lock *B* |
| | | *b.setBalance(bal*1.1)* | |
| | | *c.withdraw(bal/10)* | lock *C* |
| | | *closeTransaction* | 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* | |
|---|---|---|---|
| | | *read* | *write* |
| *Lock already set* | *none* | OK | OK |
| | *read* | OK | wait |
| | *write* | 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 *T*** | | **Transaction *U*** | |
|---|---|---|---|
| Operations | Locks | Operations | Locks |
| *a.deposit(100);* | write lock *A* | | |
| | | *b.deposit(200)* | write lock *B* |
| *b.withdraw(100)* | | | |
| ••• | waits for *U*'s | *a.withdraw(200);* | waits for *T*'s |
| | lock on *B* | ••• | lock on *A* |
| ••• | | ••• | |
| ••• | | ••• | |

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

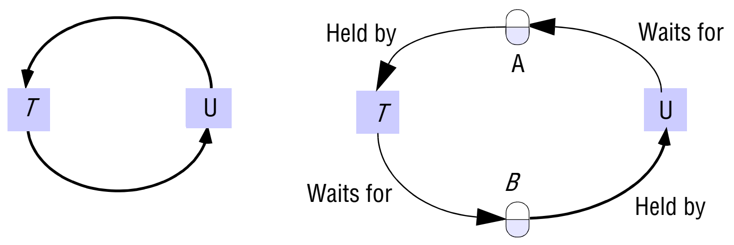
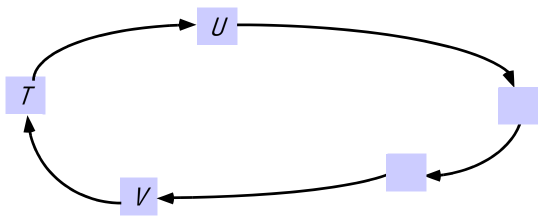


**Figure 16.21** A cycle in a wait-for graph



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.

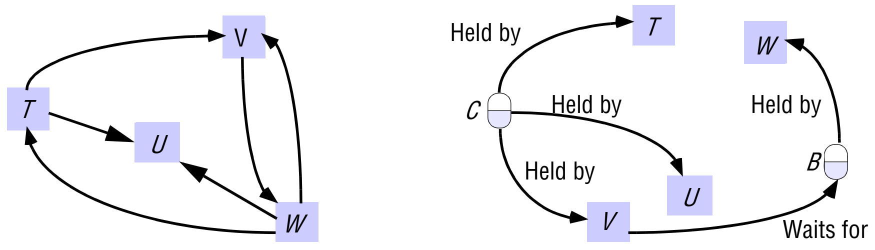
Suppose that, as in Figure 16.21, a wait-for graph contains a cycle $T \rightarrow U \rightarrow \ldots \rightarrow V \rightarrow T$. Each transaction is waiting for the next transaction in the cycle. All of these transactions are blocked waiting for locks. None of the locks can ever be released, and 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 16.22** Another wait-for graph



to avoid deadlock at this stage. Such a transaction cannot run into deadlocks with other transactions, but this approach unnecessarily restricts access to shared resources. In addition, it is sometimes impossible to predict at the start of a transaction which objects will be used. This is generally the case in interactive applications, for the user would have to say in advance exactly which objects they were planning to use – this is inconceivable in browsing-style applications, which allow users to find objects they do not know about in advance. Deadlocks can also be prevented by requesting locks on objects in a predefined order, but this can result in premature locking and a reduction in concurrency.

**Upgrade locks** • CORBA's Concurrency Control Service introduces a third type of lock, called *upgrade*, the use of which is intended to avoid deadlocks. Deadlocks are often caused by two conflicting transactions first taking read locks and then attempting to promote them to write locks. A transaction with an upgrade lock on a data item is permitted to read that data item, but this lock conflicts with any upgrade locks set by other transactions on the same data item. This type of lock cannot be set implicitly by the use of a *read* operation, but must be requested by the client.

**Deadlock detection** • Deadlocks may be detected by finding cycles in the wait-for graph. Having detected a deadlock, a transaction must be selected for abortion to break the cycle.

The software responsible for deadlock detection can be part of the lock manager. It must hold a representation of the wait-for graph so that it can check it for cycles from time to time. Edges are added to the graph and removed from the graph by the lock manager's *setLock* and *unLock* operations. At the point illustrated by Figure 16.22 on the left, it will have the following information:

| *Transaction* | *Waits for transaction* |
|---|---|
| *T* | *U*, *V* |
| *V* | *W* |
| *W* | *T*, *U*, *V* |

An edge $T \rightarrow U$ is added whenever the lock manager blocks a request by transaction *T* for a lock on an object that is already locked on behalf of transaction *U*. Note that when

**Figure 16.23** Resolution of the deadlock in Figure 16.19

| **Transaction T** | | **Transaction U** | |
|---|---|---|---|
| Operations | Locks | Operations | Locks |
| *a.deposit(100);* | write lock *A* | | |
| | | *b.deposit(200)* | write lock *B* |
| *b.withdraw(100)* | | | |
| ••• | waits for *U*'s | *a.withdraw(200);* | waits for T's |
| | lock on *B* | ••• | lock on *A* |
| | (timeout elapses) | ••• | |
| *T*'s lock on *A* becomes vulnerable, | | | |
| | unlock *A*, abort T | | |
| | | *a.withdraw(200);* | 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* | | |
|---|---|---|---|---|
| | | *read* | *write* | *commit* |
| *Lock already set* | *none* | OK | OK | OK |
| | *read* | OK | OK | wait |
| | *write* | OK | wait | – |
| | *commit* | 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.

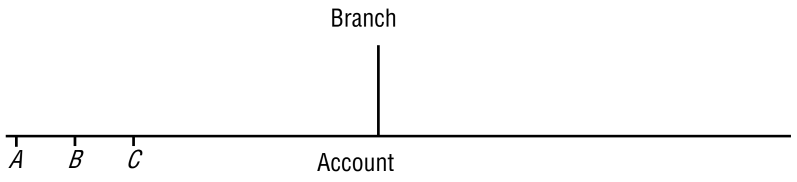
### 16.4.2 Increasing concurrency in locking schemes

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

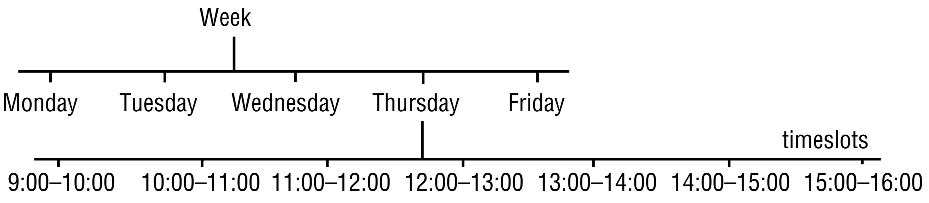


**Figure 16.27** Lock compatibility table for hierarchic locks

| *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 in *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.

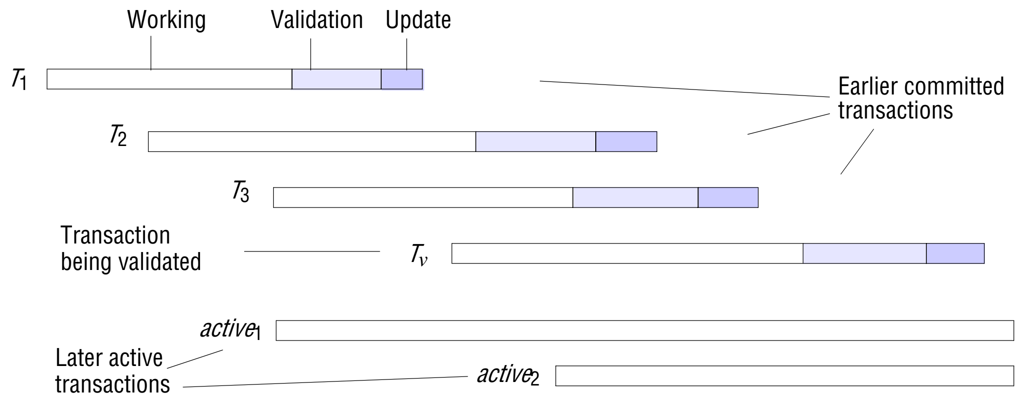
**Validation of transactions** • Validation uses the read-write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other *overlapping* transactions – that is, any transactions that had not yet committed at the time this transaction started. 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* | |
|---|---|---|---|
| *write* | *read* | 1. | $T_i$ must not read objects written by $T_v$. |
| *read* | *write* | 2. | $T_v$ must not read objects written by $T_i$. |
| *write* | *write* | 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


Working
Validation
Update
$T_1$
$T_2$
$T_3$
$T_v$
Earlier committed transactions
Transaction being validated
$active_1$
$active_2$
Later active 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 $T_i$ = $startTn+1$; $T_i$ <= $finishTn$; $T_i$++){
    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 the 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 $T_{id}$ = $active_1$; $T_{id}$ <= $active_N$; $T_{id}$++){
    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 *write* an object that has been *read* by any $T_i$ where $T_i > T_c$. This requires that $T_c \geq$ the maximum read timestamp of the object. |
| 2. | *write* | *write* | $T_c$ must not *write* an object that has been *written* by any $T_i$ where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object. |
| 3. | *read* | *write* | $T_c$ must not *read* an object that has been *written* by any $T_i$ where $T_i > T_c$. 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).

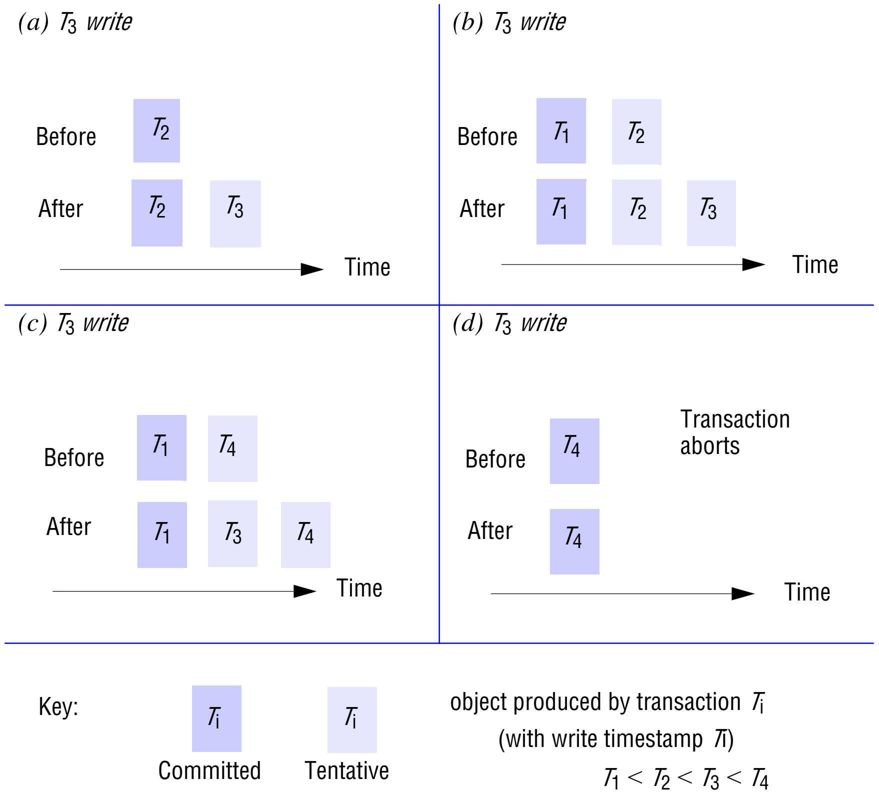
Timestamps may be assigned from the server's clock or, as in the previous section, a 'pseudo-time' may be based on a counter that is incremented whenever a timestamp value is issued. We defer until Chapter 17 the problem of generating timestamps when the transaction service is distributed and several 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 $D_{selected}$ be the version of $D$ with the maximum write timestamp ð $T_c$
    if ($D_{selected}$ is committed)
        perform read operation on the version $D_{selected}$
    else
        wait until the transaction that made version $D_{selected}$ commits or aborts
        then reapply the read rule
} else
    Abort transaction $T_c$
```

Note:

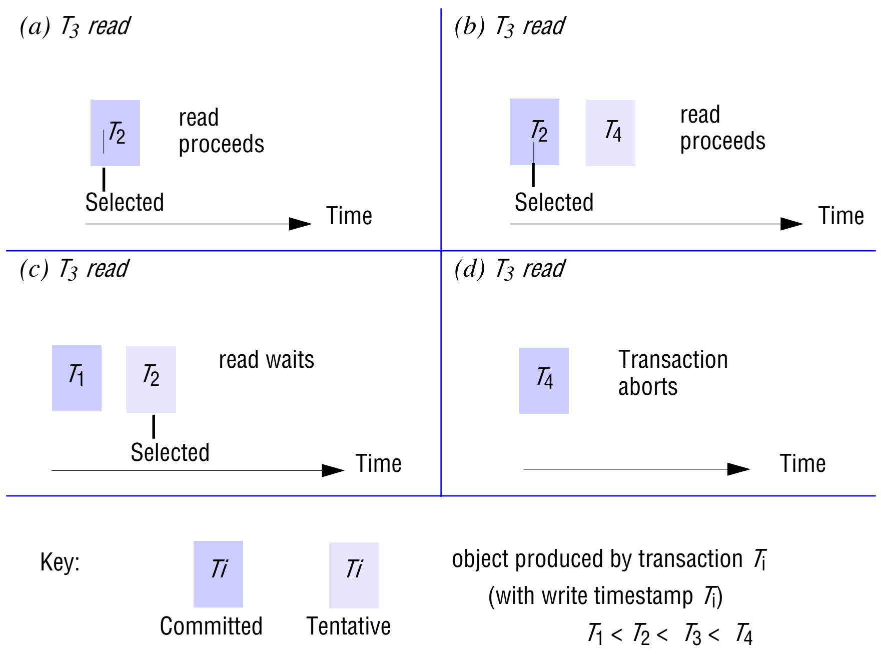
- If transaction $T_c$ has already written its own version of the object, this will be used.
- A *read* operation that arrives too early waits for the earlier transaction to complete. If the earlier transaction commits, then $T_c$ will read from its committed version. If it aborts, then $T_c$ will repeat 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)* | ••• | | ***S***, *T* | | | | |
| *commit* | ••• | | ***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.

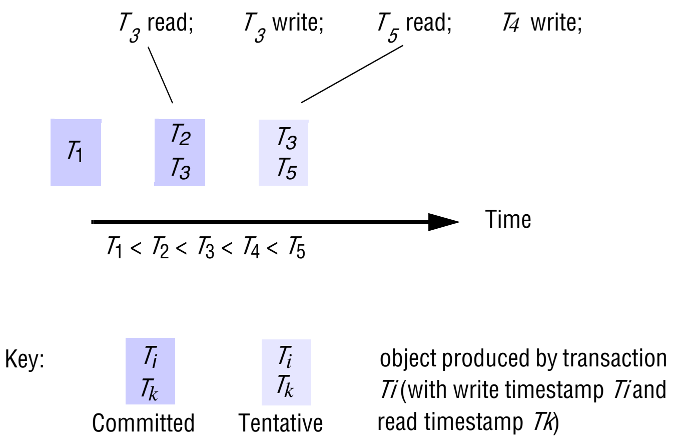
When a read arrives late, it can be allowed to read from an old committed version, so there is no need to abort late *read* operations. In multiversion timestamp ordering, *read* operations are always permitted, although they may have to *wait* for earlier transactions to complete (either commit or abort), 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.

Simililarly, 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.