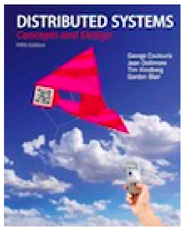

Distributed Systems: Concepts and Design

Chapter 16 Exercise Solutions

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

The TaskBag service provides the following operations:

setTask	allows clients to add task descriptions to the bag;
takeTask	allows clients to take task descriptions out of the bag.

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

(i) the server can reply immediately, telling the client to try again later;

(ii) make the server operation (and therefore the client) wait until a task becomes available;

(iii) use callbacks.

16.1 Ans.

This is a straight-forward application of the ideas on synchronising server operations. One of the projects in the Project Work section is based on the *TaskBag service*.

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

read (i) returns the value of a_i;

write(i, Value) assigns *Value* to a_i.

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

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

Give three serially equivalent interleavings of the transactions *T* and *U.*

16.2 Ans.

The interleavings of *T* and *U* are serially equivalent if they produce the same outputs (in x and y) and have the same effect on the objects as some serial execution of the two transactions. The two possible serial executions and their effects are:

If *T* runs before *U*: $x_T = a_j^0$; $y_T = a_i^0$; $x_U = a_k^0$; $y_U = 44$; $a_i = 55$; $a_j = 44$; $a_k = 66$.

If *U* runs before *T*: $x_T = a_j^0$; $y_T = 55$; $x_U = a_k^0$; $y_U = a_j^0$; $a_i = 33$; $a_j = 44$; $a_k = 66$,

where x_T and y_T are the values of x and y in transaction *T*; x_U and y_U are the values of x and y in transaction *U* and a_i^0, a_j^0 and a_k^0, are the initial values of a_i, a_j and a_k

We show two examples of serially equivalent interleavings:

A

T	*U*
x:=read(j)	
	x:= read(k)
	write(i, 55)
y:=read (i)	
	y:=read (j)
	write(k, 66)
write(j, 44)	
write(i, 33)	

B

T	*U*
x:=read(j)	
y:=read (i)	
	x:= read(k)
write(j, 44)	
write(i, 33)	
	write(i, 55)
	y:=read (j)
	write(k, 66)

A is equivalent to *U* before *T*. y_T gets the value of 55 written by *U* and at the end a_i =33; a_j =44; a_k = 66.

B is equivalent to *T* before *U*. y_U gets the value of 44 written by *T* and at the end a_i =55; a_j =44; a_k = 66.

16.3 Give serially equivalent interleaving of *T* and *U* in Exercise 16.2 with the following properties:

(i) that is strict;

(ii) that is not strict but could not produce cascading aborts;

(iii) that could produce cascading aborts.

16.3 Ans.

i) For strict executions, the *reads* and *writes* of a transaction are delayed until all transactions that have previously written the same objects are either committed or aborted. We therefore indicate the commit of the earlier transaction in our solution (a variation of B in the Answer to 13.6):

B

T	*U*
x:=read(j)	
y:=read (i)	
	x:= read(k)
write(j, 44)	
write(i, 33)	
Commit	
	write(i, 55)
	y:=read (j)
	write(k, 66)

Note that *U*'s *write(i, 55)* and *read(j)* are delayed until after *T*'s commit, because *T writes* a_i and a_j.

ii) For serially equivalent executions that are not strict but cannot produce cascading aborts, there must be no dirty reads, which requires that the *reads* of a transaction are delayed until all transactions that have previously written the same objects are either committed or aborted (we can allow *writes* to overlap). Our answer is based on B in Exercise 13.2.

B

T	*U*
x:=read(j)	
y:=read (i)	
	x:= read(k)
write(j, 44)	
write(i, 33)	
	write(i, 55)
Commit	
	y:=read (j)
	write(k, 66)

Note that *U*'s *write(i, 55)* is allowed to overlap with *T*, whereas *U*'s *read (j)* is delayed until after *T* commits.

iii) For serially equivalent executions that can produce cascading aborts, that is, dirty reads are allowed. Taking A from Exercise 13.2 and adding a *commit* immediately after the last operation of *U*, we get:

A

T	*U*
x:=read(j)	
	x:= read(k)
	write(i, 55)
y:=read (i)	
	y:=read (j)
	write(k, 66)
	commit
write(j, 44)	
write(i, 33)	

Note that *T*'s *read (i)* is a dirty read because *U* might abort before it reaches its commit operation.

16.4 The operation *create* inserts a new bank account at a branch. The transactions *T* and *U* are defined as follows:

T: aBranch.create("Z");
U: z.deposit(10); z.deposit(20).

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

T	*U*
	z.deposit(10);
aBranch.create(Z);	
	z.deposit(20);

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

16.4 Ans.

In the example, *Z*'s balance is $20 at the end.

Serial executions of *T* and *U* are:

T then *U*: *aBranch.create("Z")*; *z.deposit(10)*;*z.deposit(20)*. *Z*'s final balance is $30.

U then *T*: *z.deposit(10)*; *z.deposit(20)*; *aBranch.create("Z")*. *Z*'s final balance is $0.

Therefore the example is not a serially equivalent execution of *T* and *U*.

16.5 A newly created data item like *Z* in Exercise 16.4 is sometimes called a *phantom*. From the point of view of transaction *U*, *Z* is not there at first and then appears (like a ghost). Explain with an example, how a phantom could occur when an account is deleted.

16.5 Ans.

The point in Exercise 13.4 is that the insertion of a data item like *Z* should not be interleaved with operations on the same data item by another transaction. Suppose that we define transaction *V* as: *aBranch.delete("Z")* and consider the following interleavings of *V* and *U*:

V	*U*
	z.deposit(10);
aBranch.delete("Z")	

	z.deposit(20);

Then we have a phantom because *Z* is there at first and then disappears. It can be shown (as in Exercise 13.10) that these interleavings are not serially equivalent.

16.6 The 'Transfer' transactions *T* and *U* are defined as:

T: a.withdraw(4); b.deposit(4);

U: c.withdraw(3); b.deposit(3);

Suppose that they are structured as pairs of nested transactions:

T_1: a.withdraw(4); T_2: b.deposit(4);

U_1: c.withdraw(3); U_2: b.deposit(3);

Compare the number of serially equivalent interleavings of T_1, T_2, U_1 and U_2 with the number of serially equivalent interleavings of *T* and *U*. Explain why the use of these nested transactions generally permits a larger number of serially equivalent interleavings than non-nested ones.

16.6 Ans.

Considering the non-nested case, a serial execution with *T* before *U* is:

	T: a.withdraw(4)*;* b.deposit(4)*);*		*U*: c.withdraw(3); b.deposit(3);
T_1	a.withdraw(4);		
T_2	b.deposit(4)		
		U_1	c.withdraw(3)
		U_2	b.deposit(3)

We can derive some serially equivalent interleavings of the operations, in which *T*'s write on B must be before *U*'s read. Let us consider all the ways that we can place the operations of *U* between those of *T*.

All the interleavings must contain T_2; U_2. We consider the number of permutations of U_1 with the operations T_1-T_2 that preserve the order of *T* and *U*. This gives us (3!)/(2!*1!) = 3 serially equivalent interleavings.

We can get another 3 interleavings that are serially equivalent to an execution of *U* before *T*. They are different because they all contain U_2; T_2. The total is 6.

Now consider the nested transactions. The 4 transactions may be executed in any (serial) order, giving us 4! = 24 orders of execution.

Nested transactions allow more serially equivalent interleavings because: i) there is a larger number of serial executions, ii) there is more potential for overlap between transactions (iii) the scope of the effect of conflicting operations can be narrowed.

16.7 Consider the recovery aspects of the nested transactions defined in Exercise 16.6. Assume that a *withdraw* transaction will abort if the account will be overdrawn and that in this case the parent transaction will also abort. Describe serially equivalent interleavings of T_1, T_2, U_1 and U_2 with the following properties:

(i) that is strict;

(ii) that is not strict.

To what extent does the criterion of strictness reduce the potential concurrency gain of nested transactions?

16.7 Ans.

If a child transaction's abort can cause the parent to abort, with the effect that the other children abort, then strict executions must delay *reads* and *writes* until all the relations (siblings and ancestors) of transactions that have previously written the same objects are either committed or aborted. Our deposit and withdraw operations read and then write the balances.

i) For strict executions serially equivalent to T_1; T_2; U_1; U_2 we note that T_2 has written B. We then delay U_2's *deposit* until after the commit of T_2 and its sibling T_1. The following is an example of such an interleaving:

T_1: a.withdraw(4)	T_2: b.deposit(4);	U_1: c.withdraw(3)	U_2: b.deposit(3)
a.withdraw(3)			
	b.deposit(4)		
		c.withdraw(3)	
	commit		
commit			
			b.deposit(3)

ii) Exercise 13.6 discusses all possible serially equivalent executions. They are non-strict if they do not obey the constraints discussed in part (i).

The criterion of strictness does not in any way reduce the possible concurrency between siblings (e.g. T_1 and T_2). It does make unrelated transactions wait for entire families to commit instead of single members with which it is in conflict over access to a data item.

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

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

read (i) returns the value of a_i

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

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

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

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

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

16.8 Ans.

Because the ordering of different pairs of conflicting operations of two transactions must be the same.

For an example where locks are released early:

T	*T's locks*	*U*	*U's locks*
	lock i		
x:= read (i);			
	unlock i		
			lock i
		write(i, 55);	
			lock j
		write(j, 66);	
		commit	unlock i, j
	lock j		
write(j, 44);			
	unlock j		
commit			

T conflicts with *U* in access to a_i. Order of access is *T* then *U*.

T conflicts with *U* in access to a_j. Order of access is *U* then *T*. These interleavings are not serially equivalent.

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

T: x= read (i); write(j, 44);
U: write(i, 55);write(j, 66);

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

(a)

T	*U*
x= read (i);	
	write(i, 55);
write(j, 44);	
	write(j, 66);

(b)

T	*U*
x= read (i);	
write(j, 44);	
	write(i, 55);
	write(j, 66);

(c)

T	*U*
	write(i, 55);
	write(j, 66);
x= read (i);	
write(j, 44);	

(d)

T	*U*
	write(i, 55);
x= read (i);	
	write(j, 66);
write(j, 44);	

16.9 Ans.

a) serially equivalent but not with two-phase locking.
b) serially equivalent and with two-phase locking.
c) serially equivalent and with two-phase locking.
d) serially equivalent but not with two-phase locking.

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

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

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

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

16.10 Ans.

There is no guarantee of consistent retrievals because overlapping transactions can alter the objects after they are unlocked.

The database does not become inconsistent.

T	*T's locks*	*U*	*U's locks*
	lock i		
x:= read (i);			
	unlock i		
			lock i
		write(i, 55)	
			lock j
		write(j, 66)	
		Commit	unlock i, j
	lock j		
y:= read(j)			
Commit	unlock j		

In the above example *T* is read only and conflicts with *U* in access to a_i and a_j. a_i is accessed by *T* before *U* and a_j by *U* before *T*. The interleavings are not serially equivalent. The values observed by *T* are x=10, y= 66, and the values of the objects at the end are a_i=55, a_j= 66.

Serial executions give either (*T* before *U*) x=10, y=20, a_i=55, a_j=66; or (*U* before *T*) x=55, y=66, a_i=55, a_j=66). This confirms that retrievals are inconsistent but that the database does not become inconsistent.

16.11 The executions of transactions are strict if *read* and *write* operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted. Explain how the locking rules in Figure 16.16 ensure strict executions.

16.11 Ans.

If a previous transaction has written a data item, it holds its locks until after it has committed, therefore no other transaction may either *read* or *write* that data item (which is the requirement for serial executions).

16.12 Describe how a non-recoverable situation could arise if write locks are released after the last operation of a transaction but before its commitment.

16.12 Ans.

An earlier transaction may release its locks but not commit, meanwhile a later transaction uses the objects and commits. Then the earlier transaction may abort. The later transaction has done a dirty read and cannot be recovered because it has already committed.

16.13 Explain why executions are always strict even if read locks are released after the last operation of a transaction but before its commitment. Give an improved statement of Rule 2 in Figure 16.16.

16.13 Ans.

Strict executions require that *read* and *write* operations on a data item are delayed until all transactions that previously wrote that data item have either committed or aborted. The holding of a write lock is sufficient to protect future transactions from non-strictness because we are concerned only with previous *write* operations.

Rule 2: When the client indicates that the last operation has been done (by a request to commit or abort), release read locks. Hold write locks until commit or abort is completed.

16.14 Consider a deadlock detection scheme for a single server. Describe precisely when edges are added to and removed from the wait-for-graph.

Illustrate your answer with respect to the following transactions *T*, *U* and *V* at the server of Exercise 16.8.

T	*U*	*V*
	write(i, 66)	
write(i, 55)		
		write(i, 77)
	commit	

When *U* releases its write lock on a_i, both *T* and *V* are waiting to obtain write locks on it. Does your scheme work correctly if *T* (first come) is granted the lock before *V*? If your answer is 'No', then modify your description.

16.14 Ans.

Scheme:

When transaction *T* blocks on waiting for transaction *U*, add edge $T \rightarrow U$

When transaction *T* releases a lock, remove all edges leading to *T*.

Illustration: *U* has write lock on a_i.

T requests write a_i. Add $T \rightarrow U$

V requests write a_i. Add $V \rightarrow U$

U releases a_i. Delete both of above edges.

No it does not work correctly! When *T* proceeds, the graph is wrong because *V* is waiting for *T* and it should indicate $V \rightarrow$T.

Modification: we could make the algorithm unfair by always releasing the last transaction in the queue.

To make it fair: store both direct and indirect edges when conflicts arise. In our example, when transaction *T* blocks on waiting for transaction *U* add edge $T \rightarrow U$ then, when *V* starts waiting add $V \rightarrow U$ and $V \rightarrow T$

16.15 Consider hierarchic locks as illustrated in Figure 16.26. What locks must be set when an appointment is assigned to a time-slot in week *w*, day *d*, at time, *t*? In what order should these locks be set? Does the order in which they are released matter?

What locks must be set when the time slots for every day in week **w** are viewed?

Can this be done when the locks for assigning an appointment to a time-slot are already set?

16.15 Ans.

Set write lock on the time-slot *t*, intention-to-write locks on week *w* and day *d* in week *w*. The locks should be set from the top downwards (i.e. week *w* then day *d* then time *t*). The order in which locks are released does matter - they should be released from the bottom up.

When week *w* is viewed as a whole, a read lock should be set on week *w*. An intention-to-write lock is already set on week *w* (for assigning an appointment), the read lock must wait (see Figure 13.27).

16.16 Consider optimistic concurrency control as applied to the transactions *T* and *U* defined in Exercise 16.9. Suppose that transactions *T* and *U* are active at the same time as one another. Describe the outcome in each of the following cases:

i) *T*'s request to commit comes first and backward validation is used;

ii) *U*'s request to commit comes first and backward validation is used;

iii) *T*'s request to commit comes first and forward validation is used;

iv) *U*'s request to commit comes first and forward validation is used.

In each case describe the sequence in which the operations of *T* and *U* are performed, remembering that writes are not carried out until after validation.

16.16 Ans.

i) *T*'s *read(i)* is compared with *write*s of overlapping committed transactions: OK (*U* has not yet committed).

U - no *read* operations: OK.

ii) *U* - no *read* operations: OK.

T's *read(i)* is compared with *write*s of overlapping committed transactions (*U*'s *write(i)):* FAILS.

iii) *T*'s *write(j)* is compared with *reads* of overlapping active transactions (*U*): OK.

U's *write(i)* is compared with *reads* of overlapping active transactions (none): OK (*T* is no longer active).

iv) *U*'s *write(i)* is compared with *reads* of overlapping active transactions (*T*'s *read(i)*): FAILS.

T's *write(j)* is compared with *reads* of overlapping active transactions (none): OK.

(i)

T	*U*
x:= read (i);	
write(j, 44);	
	write(i, 55);
	write(j, 66);

(ii)

T	*U*
	write(i, 55);
x:= read (i);	*write(j, 66);*
Abort	

(iii)

T	*U*
x:= read (i);	
write(j, 44);	
	write(i, 55);
	write(j, 66);

(iv)

T	*U*
x:= read (i);	
	Abort
write(j, 44);	

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

T	*U*
openTransaction	*openTransaction*
y= read(k);	
	write(i, 55);
	write(j, 66);
	commit
x= read(i);	
write(j, 44);	

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

16.17 Ans.

Keep ordered read sets for active transactions. When a transaction commits after passing its validation, note the fact in the read sets of all active transactions. For example, when *U* commits, note the fact in *T*'s read set.

Thus, *T*'s read set = {*U* commit, i}

Then the new validate procedure becomes:

```
  boolean valid = true
for ($T_i$ = startTn + 1; $T_i$++; $T_i$ <= finishTn) {
    let S = set of members of read set of $T_j$ before commit $T_i$
    IF S intersects write set of $T_1$
        THEN valid := false
}
```

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

16.18 Ans.

The order of interleavings allowed with two-phase locking depends on the order in which T and U access a_i. If T is first we get (b) and if U is first we get (c) in Exercise 13.9.

The ordering of 13.9b for two-phase locking is the same as 13.16 (i) optimistic concurrency control.

The ordering of 13.9c for two-phase locking is the same as 13.16 (ii) optimistic concurrency control if we allow transaction T to restart after aborting.

In this example, the sequences of operations are the same for both methods.

.

16.19 Consider the use of timestamp ordering with each of the example interleavings of transactions T and U in Exercise 16.9. Initial values of a_i and a_j are 10 and 20, respectively, and initial read and write timestamps are t_0. Assume each transaction opens and obtains a timestamp just before its first operation, for example, in (a) T and U get timestamps t_1 and t_2 respectively where $0 < t_1 < t_2$. Describe in order of increasing time the effects of each operation of T and U. For each operation, state the following:

i) whether the operation may proceed according to the write or read rule;

ii) timestamps assigned to transactions or objects;

iii) creation of tentative objects and their values.

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

16.19 Ans.

a) Initially:

a_i: value = 10; write timestamp =max read timestamp = t_0
a_j: value = 20; write timestamp =max read timestamp = t_0

T: *x:= read (i)*; T timestamp = t_1;

read rule: t_1> write timestamp on committed version (t_0) and D_{selected} is committed:
allows *read* x = 10; max read timestamp(a_i) = t_1. (see Figure 13.31a and read rule page 500)

U: *write(i, 55)*; U timestamp = t_2

write rule: t_2 >= max read timestamp (t_1) and t_2> write timestamp on committed version (t_0):
allows *write* on tentative version a_i: value = 55; write timestamp =t. (See write rule page 499)

T: *write(j, 44)*;

write rule: t_1 >= max read timestamp (t_0) and t_1 > write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 44; write timestamp(a_j) =t_1

U: *write(j, 66)*;

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_0)
allows *write* on tentative version a_j: value = 66; write timestamp =t_2

T commits first:

a_j: committed version: value = 44; write timestamp =t_1; read timestamp = t_0

U commits:

a_i: committed version: value = 55; write timestamp $=t_2$; max read timestamp = t_1
a_j: committed version: value = 66; write timestamp $=t_2$; max read timestamp = t_0

b) Initially as (a);

T: *x:= read (i)*; *T* timestamp = t_1;

read rule: $t_1 >$ write timestamp on committed version (t_0) and D_{selected} is committed:
allows *read,* x = 10; max read timestamp(a_i) = t_1

T: *write(j,44)*

write rule: $t_1 >=$ max read timestamp (t_0) and $t_1 >$ write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 44; write timestamp $=t_1$

U: *write(i, 55*); *U* timestamp = t_2

write rule: $t_2 >=$ max read timestamp (t_1) and $t_2 >$ write timestamp on committed version (t_0):
allows *write* on tentative version a_i: value = 55; write timestamp $=t_2$

U: *write(j, 66)*;

write rule: $t_2 >=$ max read timestamp (t_0) and $t_2 >$ write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 66; write timestamp $=t_2$

T commits first:

a_j: committed version: value = 44; write timestamp $=t_1$; max read timestamp = t_0

U commits:

a_i: committed version: value = 55; write timestamp $=t_2$; max read timestamp = t_1
a_j: committed version: value = 66; write timestamp $=t_2$; max read timestamp = t_0

c) Initially as (a);

U: *write(i, 55*); *U* time stamp = t_1;

write rule: $t_1 >=$ max read timestamp (t_0) and $t_2 >$ write timestamp on committed version (t_0):
allows *write* on tentative version a_i: value = 55; write timestamp $=t_1$

U: *write(j, 66)*;

write rule: $t_2 >=$ max read timestamp (t_0) and $t_2 >$ write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 66; write timestamp $=t_1$

T: *x:= read (i)*; *T* time stamp = t_2

read rule: $t_2 >$ write timestamp on committed version (t_0), write timestamp of $D_{\text{selected}} = t_1$ and D_{selected} is not committed: WAIT for *U* to commit or abort. (See Figure 13.31 c)

U commits:

a_i: committed version: value = 55; write timestamp $=t_1$; max read timestamp(a_i) = t_0
a_j: committed version: value = 66; write timestamp $=t_1$; max read timestamp(a_j) = t_0

T continues by reading version made by *U* x = 55; write timestamp(a_i) $=t_1$; read timestamp(a_i) = $t_2 t_2$

T: *write(j,44)*

write rule: $t_2 >=$ max read timestamp (t_0) and $t_2 >$ write timestamp on committed version (t_1):
allows *write* on tentative version a_j: value = 44; write timestamp $=t_2$

T commits:

a_i: committed version: value = 55; write timestamp $=t_1$; max read timestamp = t_2

a_j: committed version: value =44; write timestamp $=t_2$; max read timestamp = t_0

d) Initially as (a);

U: *write(i, 55*); *U* time stamp = t_1;

write rule: $t_1 >=$ max read timestamp (t_0) and $t_2 >$ write timestamp on committed version (t_0)
allows *write* on tentative version a_i: value = 55; write timestamp(a_i) $=t_1$

T: *x:= read (i)*; *T* time stamp = t_2

read rule: t_2> write timestamp on committed version (t_0), write timestamp of $D_{selected}$ = t1 and $D_{selected}$ is not committed: Wait for *U* to commit or abort. (See Figure 13.31 c)

U: *write(j, 66)*;

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version(t_0) allows *write* on tentative version a_j: value = 66; write timestamp =t_1

U commits:

a_i: committed version: value = 55; write timestamp =t_1; max read timestamp = t_0
a_j: committed version: value = 66; write timestamp =t_1; max read timestamp = t_0

T continues by reading version made by *U*, x = 55; max read timestamp(a_i) = t_2

T: *write(j,44)*

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_1) allows *write* on tentative version a_j: value = 44; write timestamp=t_2

T commits:

a_i: committed version: value = 55; write timestamp =t_1; max read timestamp = t_2
a_j: committed version: value = 44; write timestamp =t_2; max read timestamp = t_0

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

T	*U*
openTransaction	
	openTransaction
	write(i, 55);
	write(j, 66);
x= read (i);	
write(j, 44);	
	commit

T	*U*
openTransaction	
	openTransaction
	write(i, 55);
	write(j, 66);
	commit
x= read (i);	
write(j, 44);	

16.20 Ans.

The difference between this exercise and the interleavings for Exercise 13.9(c) is that *T* gets its timestamp before *U*. The difference between the two orderings in this exercise is the time of the commit requests. Let t_1 and t_2 be the timestamps of *T* and *U*. The initial situation is as for 13.9 (a).

U: *write(i, 55)*;

write rule: t_2 >= max read timestamp (t_0) and t_2> write timestamp on committed version (t_0) allows *write* on tentative version a_i: value = 55; write timestamp =t_2

U: *write(j, 66)*;

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_0) allows *write* on tentative version a_j: value = 66; write timestamp =t_2

T: *x:= read (i)*;

read rule: t_1> write timestamp t_0 on committed version and $D_{selected}$ is committed: allows *read,* x = 10; max read timestamp(a_i) = t_1

T: *write(j,44)*

write rule: t_1 >= max read timestamp (t_1) and t_1 > write timestamp on committed version (t_0) allows *write* on tentative version a_j: value = 44; write timestamp =t_2

U commits:

a_i: committed version: value = 55; write timestamp =t_2; max read timestamp = t_1

a_j: *T* has a tentative version with write timestamp = t_1. *U*'s version with value = 66 and write timestamp =t_2 cannot be committed until after *T*'s version.

T commits:

a_j: committed version: value = 44; write timestamp =t_1; max read timestamp = t_0; *T*'s version is replaced with *U*'s version: value = 66; write timestamp =t_2;   max read timestamp = t_0

The second ordering proceeds in the same way as the first until *U* has performed both its *write* operations and commits. At this stage we have

a_i: committed version: value = 55; write timestamp =t_2; max read timestamp = t_0

a_j: committed version: value = 66; write timestamp =t_2;   max read timestamp = t_0

T: *x:= read (i)*;

read rule: NOT t_1> write timestamp t_2 on committed version
T is Aborted (see Figure 13.31 d)

16.21 Repeat Exercise 16.20 using multiversion timestamp ordering.
16.21 Ans.

The main difference for multiversion timestamp ordering is that *read* operations can use old committed versions of objects instead of aborting when they are too late (see Page 502). The read rule is:

```
let $D_{\text{selected}}$ be the version of D with the maximum write timestamp <= Tj
IF $D_{\text{selected}}$ is committed THEN
    perform read operation on the version $D_{\text{selected}}$
ELSE Wait until the transaction that made version $D_{\text{selected}}$ commits or aborts
```

write operations cannot be too late, but writes are checked against potentially conflicting *read* operations. The write rule is taken from page 402. Recall that each data item has a history of committed versions.

For the ordering on the left of Exercise 13.20, we show that the outcome is the same. Timestamps are *T*: t_1 and *U*:t_2. Initial state as in Exercise 13.9 a. Call these committed versions V_{t0}.

U: *write(i, 55*);

write rule: read timestamp of $D_{\text{maxEarlier}}$ t_0 <=t_2
allows *write* on tentative version a_i: value = 55; write timestamp =t_2

U: *write(j, 66)*;

write rule: read timestamp of $D_{\text{maxEarlier}}$ t_0 <= t_2
allows *write* on tentative version a_j: value = 66; write timestamp =t_2

T: *x:= read (i)*;

Select version with write timestamp t_0 *read* x = 10; its read timestamp becomes t_1

T: *write(j,44)*

write rule: read timestamp of $D_{\text{maxEarlier}}$ t_1 <= t_1
allows *write* on tentative version a_j: value = 44; write timestamp =t_1

U commits:

a_i: committed version V_{t2}: value = 55; write timestamp =t_2;
a_j: committed version V_{t2}: value = 66 and write timestamp =t_2

T commits:

a_j: committed version V_{t1}: value = 44; write timestamp =t_1

For the ordering on the right of Exercise 13.20, we show that the outcome is different in that *T* does not abort. It proceeds in the same way as the first until *U* has performed both its *write* operations and commits. At this stage we have:

a_i: committed version V_{t2}: value = 55; write timestamp =t_2
a_j: committed version V_{t2}: value = 66; write timestamp =t_2

T: *x:= read (i)*;

The *read* selects the committed version V_{t0} and gets x=10 and the read timestamp = t_1

T: *write(j,44)*

write rule: read timestamp of $D_{\text{maxEarlier}}$ t_1 <= t1
allows *write* on tentative version. a_j: value = 44; write timestamp =t_2

T commit:

a_j: committed version V_{t1}: value = 44; write timestamp =t1

16.22 In multiversion timestamp ordering, *read* operations can access tentative versions of objects. Give an example to show how cascading aborts can happen if all read operations are allowed to proceed immediately.

16.22 Ans.

The answer to Exercise 13.21 gives the read rule for multiversion timestamp ordering. *read* operations are delayed to ensure recoverability. In fact this delay also prevents dirty reads and cascading aborts.

Suppose now that *read* operations are not delayed, but that commits are delayed as follows to ensure recoverability. If a transaction, *T* has observed one of *U*'s tentative objects, then *T*'s commit is delayed until *U* commits or aborts (see page 503). Cascading aborts can occur when *U* aborts because *T* has done a dirty read and will have to abort as well.

To find an example, look for an answer to Exercise 13.19 where a *read* operation was delayed (i.e. (c) or (d)). Consider the diagram for (c) in Exercise 13.2. With delays, *T*'s *x := read(i)* is delayed until *U* commits or aborts. Now consider allowing *T*'s *x := read(i)* to use *U*'s tentative version immediately. Then consider the situation in which *T* asks to commit and *U* subsequently aborts. Note that *T*'s commit request is delayed until the outcome of *U* is known, so the situation is recoverable, but *T* has performed a 'dirty read' and must be aborted. Cascading aborts can now arise because some other transactions may have observed *T*'s tentative objects.

16.23 What are the advantages and drawbacks of multiversion timestamp ordering in comparison with ordinary timestamp ordering?

16.23 Ans.

The algorithm allows more concurrency than single version timestamp ordering but incurs additional storage costs.

Advantages:

The presence of multiple committed versions allows late *read* operations to succeed.

write operations are allowed to proceed immediately unless they will invalidate earlier reads (a *write* by a transaction with timestamp T_i is rejected if a transaction with timestamp T_j has read a data item with write timestamp T_k and $T_k < T_i < T_j$).

Drawbacks:

The algorithm requires storage for multiple versions of each committed objects and for information about the read and write timestamps of each version to be used in carrying out the read and write rules. In the case that a version is deleted, *read* operations will have to be rejected and transactions aborted.

Exercise 13.22 shows that the algorithm can provide yet more concurrency, at the risk of cascading aborts, by allowing *read* operations to proceed immediately. In this case, to ensure recoverability, requests to commit must be delayed until any the completion (commitment or abortion) of any transaction whose tentative objects have been observed.