CSE 486/586 Non-Graded Practice Problem Set 2 Solutions

Acknowledgement: This problem set heavily contains the material developed and copyrighted by Prof. Indranil Gupta at Illinois.

1. You are designing a new kind of electronic voting machine. These machines (i.e., clients) will report each vote made at them to the server via an RPC. Discuss the pros and cons of using each of the following semantics for the RPC in this system: (a) maybe, (b) at least once, (c) at most once. Which would you recommend and why?

A: At-most-once semantics using a reliable transport. At-least-once semantics is also acceptable if it is made clear that the server maintains a log of all previous requests and eliminates any duplicates.

2. Problem 16.2

A: 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: xT = aj0; yT = ai0; xU = ak0; yU= 44; ai* =55; *aj* =44; *ak* = 66.
If *U* runs before *T: xT = aj0; yT = 55; xU = ak0; yU= aj0; ai* =33; *aj* =44; *ak* = 66,

where *xT* and *yT* are the values of x and y in transaction *T; xU* and *yU* are the values of x and y in transaction *U* and *ai0, aj0* and *ak0,* are the initial values of *ai, aj* and *ak.*

We show two examples of serially equivalent interleavings:

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

This equivalent to *U* before *T. yT* gets the value of 55 written by *U* and at the end *ai* =33; *aj* =44; *ak* = 66.

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

This is equivalent to *T* before *U. yU* gets the value of 44 written by *T* and at the end *ai* =55; *aj* =44; *ak* = 66.

3. Problem 16.3

A:

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:

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

U: y = read (j); U: write(k, 66);

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

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 the second example in Exercise 16.2.

T: x= read (j); T: y = read (i); U: x= read(k); T: write(j, 44); T: write(i, 33); U: write(i, 55); T: Commit; U: y = read (j); U: 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 the first example from Exercise 16.2 and adding a *commit* immediately after the last operation of *U,* we get:

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

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

4. Problem 16.10

A: 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 *ai* and *aj. ai* is accessed by *T* before *U* and *aj* 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 *ai=55, aj=* 66.

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

5. Problem 16.12

A: 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.

6. Problem 16.14

A:

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 *ai.*

T requests write *ai.* Add $T \rightarrow U$
V requests write *ai.* Add $V \rightarrow U$
U releases *ai.* Delete both of above edges.

This 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 $V \rightarrow T$

7. Problem 17.1

A: In both cases, we consider the normal case with no time outs. In the decentralised version of the two-phase commit protocol:

No of messages:

Phase 1: coordinator sends its vote to *N* workers = *N*
Phase 2: each of *N* workers sends its vote to (N-1) other workers + coordinator = *N*(N* - 1).
Total = *N*N.*

No. of rounds:

coordinator to workers + workers to others = 2 rounds.

Advantages: the number of rounds is less than for normal two-phase commit protocol which requires 3. Disadvantages: the number of messages is far more: *N*N* instead of 3N.

8. Problem 17.2

A: In the two-phase commit protocol: the ‘uncertain’ period occurs because a worker has voted *yes* but has not yet been told the outcome. (It can no longer abort unilaterally).

In the three-phase commit protocol: the workers ‘uncertain’ period lasts from when the worker votes *yes* until it receives the *PreCommit* request. At this stage, no other participant can have committed. Therefore if a group of workers discover that they are all ‘uncertain’ and the coordinator cannot be contacted, they can decide unilaterally to abort.

9. Problem 17.4

A:

Schedule at server *X:*
T: Read(A); Write(A); *U:Read(A);* Write(A); serially equivalent with *T* before *U*

Schedule at Server *Y:*
U: Read(B); Write(B); *T:* Read(B); Write(B); serially equivalent with *U* before *T*

This is not serially equivalent globally because there is a cycle $T \rightarrow U \rightarrow T.$

10. Problem 18.4

A: Process *p* must receive a new group view containing only itself, and it must receive the message it sent. The question is: in what order should these events be delivered to *p?* If *p* received the message first, then that would tell *p* that *q* and *r* received the message; but the question implies that they did not receive it. So *p* must receive the group view first.

11. Problem 18.5

A: If the multicast is reliable, yes. Then we can solve consensus. In particular, we can decide, for each message, the view of the group to deliver it to. Since both messages and new group views can be totally ordered, the resultant communication will be view-synchronous. If the multicast is unreliable, then we do not have a way of ensuring the consistency of view delivery to all of the processes involved.

12. Problem 18.6

A: Sync-ordering the remove-user update ensures that all processes handle the same set of operations on a thingumajig before the user is removed. If removal were only causally ordered, there would not be any definite delivery ordering between that operation and any other on the thingumajig. Two processes might receive an operation from that user respectively before and after the user was removed, so that one process would reject the operation and the other would not.

13. Problem 18.9

A: See pp. 776-778 for the difference. In the absence of clock synchronization of sufficient precision, it is pretty much the case that linearizability can only be achieved by funnelling all requests through a single server.

14. Problem 18.10

A: Due to delays in update propagation, a *read* operation processed at a backup could retrieve results that are older than those at the primary – that is, results that are older than those of an earlier operation requested by another process. So the execution is not linearizable.

The system is sequentially consistent, however: the primary totally orders all updates, and each process sees some consistent interleaving of *reads* between the same series of updates.

15. Problem 18.18

A: An interleaving of T and U at the replicas assuming that two-phase locks are applied to the replicas:

T		U	
x: read(Ax)	lock Ax		
write(Bm, 44)	lock Bm		
		x := read(Bm)	Wait
write(Bn, 44)	lock Bn	.	
Commit unlock	Ax,Bm,Bn	.	
		write(Ax, 55)	lock Ax
		write(Ay, 55)	lock Ay

Suppose Bm fails before T locks it, then U will not be delayed. (It will get a lost update). The problem arises because Read can use one of the copies before it fails and then Write can use the other copy. Local validation ensures one copy serializability by checking before it commits that any copy that failed has not yet been recovered. In the case of T, which observed the failure of Bm, Bm should not yet have been recovered, but it has, so T is aborted.

16. Problem 18.19

A:

i) Partition separates X and Y from Z when all data items have version v0 say:

X	Y	Z
A = 100 (vo)	A = 100 (vo)	A = 100 (vo)
B = 100 (vo)	B = 100 (vo)	B = 100 (vo)

A client reads the value of A and then writes it to B:
read quorum = 1+1 for A and B - client Reads A from X or Y
write quorum = 1+1 for B client Writes B at X and Y

ii) Client can access only server Z: read quorum = 1, so client cannot read, write quorum = 1 so client cannot write, therefore neither operation takes place.

iii) After the partition is repaired, the values of A and B at Z may be out of date, due to clients having written new values at servers X and Y. e.g. versions v1:

X	Y	Z
A = 200 (v1)	A = 200 (v1)	A = 100 (vo)

B = 300 (v1)	B = 300 (v1)	B = 100 (vo)

Then another partition occurs so that X and Z are separated from Y.

The client *Read* request causes an attempt to obtain a read quorum from X and Z. This notes that the versions (v0) at Z are out of date and then Z gets up-to-date versions of A and B from X.

Now the read quorum = 1+1 and the read operation can be done. Similarly the write operation can be done.

17. Explain why the following sequence is allowed for a causally-consistent storage, but not with a sequentially consistent storage. Note that W(x)a denotes a write operation on variable x that writes value a and R(x)a denotes a read operation on variable x that returns a. There are 4 processes, P1 - P4.

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

A: In a causally-consistent storage, the processes only need to see causally-related writes in the same order. In the above scenario, P1's W(x)a and P2's W(x)b are causally-related and all processes see value a first, then value b next. Thus, it is allowed for a causally-consistent storage.

On the other hand, in a sequentially-consistent storage, the processes need to see all writes in the same order that preserves each program order. In the above scenario, P3 sees value c first, then value b next, but P4 sees value b first, then value c next. A sequentially-consistent storage should not allow that ordering.

18. Suppose there are three processes, among which one process is faulty. Using Paxos, describe the steps the three processes can take in order to agree on a value.

A: Please refer to the Paxos protocol.

19. Problem 15.23

A:

Any lieutenant can verify the signature on any message. No lieutenant can forge another signature. The correct lieutenants sign what they each received and send it to one another.

A correct lieutenant decides *x* if it receives messages [x](signed commander) and either [[x](signed commander)](signed lieutenant) or a message that either has a spoiled lieutenant signature or a spoiled commander signature.

Otherwise, it decides on a default course of action (retreat, say).

A correct lieutenant either sees the proper commander's signature on two different courses of action (in which case both correct lieutenants decide 'retreat'); or, it sees one good signature direct from the commander and one improper commander signature (in which case it decides on whatever the commander signed to do); or it sees no good commander signature (in which case both correct lieutenants decide 'retreat').

In the middle case, either the commander sent an improperly signed statement to the other lieutenant, or the other lieutenant is faulty and is pretending that it received an improper signature. In the former case, both correct lieutenants will do whatever the (albeit faulty) commander told one of them to do in a signed message. In the latter case, the correct lieutenant does what the correct commander told it to do.

20. (a) Consider a PBFT system with 2 faulty nodes. What is the minimum number of non-faulty nodes required in the system to support Byzantine fault tolerance? (b) In this system, a client issues a request to the primary and the primary starts the three phase protocol (you can assume that the primary is non-faulty). Calculate the total number of messages exchanged by the non-faulty replicas until all of them reply to the client. It is given that the system only supports unicast messages.

A:

(a) Minimum number of non-faulty nodes = 2f + 1 = 5
(b) Pre-prepare phase messages (primary to all the replicas) = 6
Prepare phase messages (Non-faulty replicas, except the primary, to all other replicas) =
4 × 6 = 24
Commit phase messages (All non-faulty replicas to all other replicas) = 5 × 6 = 30
Total messages = 6 + 24 + 30 = 60