Homework 4

CS425/ECE428 Spring 2023

Due: Monday, April 10 at 11:59 p.m.

1. **Transaction Processing**

   A bank uses a transaction processing system that complies with ACID properties. Within each transaction, a user can issue one or more of the following operations: (i) `DEPOSIT <account> <amount>` which deposits the specified amount into the specified account, (ii) `WITHDRAW <account> <amount>` which withdraws the specified amount from the specified account, and (iii) `BALANCE <account>` which immediately displays the current balance in the specified account (also including the effects of operations previously executed within the same transaction). As a consistency check, if at the *end of a transaction*, any account has a negative balance, the system aborts that transaction.

   Consider the five transactions shown below that are executed serially (one after another) in order T1, T2, T3, T4, T5. Answer the following questions, assuming all accounts referred in the transactions have a balance of zero before T1 is executed.

   `T1: DEPOSIT A 20; DEPOSIT B 50; DEPOSIT C 40`

   `T2: DEPOSIT A 30; WITHDRAW B 60; WITHDRAW C 10; BALANCE C`

   `T3: DEPOSIT A 10; WITHDRAW B 40; WITHDRAW C 10; BALANCE B`

   `T4: WITHDRAW A 50; DEPOSIT B 10; WITHDRAW C 20`

   `T5: BALANCE A; BALANCE B; BALANCE C`

   (a) (5 points) For each transaction, state whether it gets committed or aborted and why.

   > **Solution:** T1 will get committed since at the end of transaction all accounts have positive balance.
   > T2 will get aborted since `WITHDRAW B 60` will result in negative balance.
   > T3 will get committed since at the end of transaction all accounts have positive balance.
   > T4 will get aborted since `WITHDRAW A 50` will result in negative balance.
   > T5 will get committed since all operations are only asking for the balance of each account.

   (b) (5 points) What will be the result displayed by each of the `BALANCE` operations invoked in the transactions?

   > **Solution:** T2 `BALANCE C`: 30
   > T3 `BALANCE B`: 10
   > T5 `BALANCE A`: 30
   > T5 `BALANCE B`: 10
   > T5 `BALANCE C`: 30

2. **Concurrency: Two-phase locking, Deadlocks, and Timestamped Ordering**

   Consider the following two transactions, each with five operations:

	$T1$	$T2$
1	write E	write A
2	read D	write B
3	read B	write C
4	read A	read B
5	write C	read D

   (a) (2 points) Write down all the conflicting pairs of operations across the two transactions. (You can refer to each operation as $Tn.m$; e.g., $T2.1$ is "write A", $T2.2$ is "write B", and so on).

Solution: $T1.3$ and $T2.2$
$T1.4$ and $T2.1$
$T1.5$ and $T2.3$

(b) (3 points) Is the following interleaving of operations across $T1$ and $T2$ serially equivalent? Explain why or why not.

$T1$	$T2$
	write A
write E	
read D	
read B	
	write B
	write C
	read B
read A	
	read D
write C	

Solution: They are not serially equivalent since $T1$'s read B and $T2$'s write B is a conflicting pair with order $T1, T2$, but $T1$'s read A and $T2$'s write A is another conflicting pair with order $T2, T1$.

(c) (4 points) Is there a *non-serial* interleaving of operations across $T1$ and $T2$, that could result from using strict two-phase locking (with read-write locks), and is equivalent in effect to a serial execution of $T1$ followed by $T2$? If yes, provide an example. If not, explain why.

Solution: This is impossible since $T1$ will hold read_lock(A) until it completes the last operation, but $T2$'s first operation requires write_lock(A). Thus, $T2$'s first operation will always be after $T1$'s last operation, which means it will always be a serial interleaving of operations across $T1$ and $T2$.

(d) (4 points) Is there a *non-serial* interleaving of operations across $T1$ and $T2$, that could result from using strict two-phase locking (with read-write locks), and is equivalent in effect to a serial execution of $T2$ followed by $T1$? If yes, provide an example. If not, explain why.

Solution: One possible solution:

$T1$	$T2$
	write A
	write B
	write C
write E	
read D	
	read B
	read D
read B	
read A	
write C	

(e) (4 points) Write down a partial interleaving of the operations across $T1$ and $T2$ that is compliant with strict two-phase locking (with read-write locks) and leads to a deadlock. List which lock (and in which mode – read or write) will be waited upon by each transaction in your deadlock.

Solution: One possible solution:

$T1$	$T2$
write E	
read D	
read_lock(B)	
read B	
	write_lock(A)
	write A
	wait for write_lock(B)
	write B
wait for read_lock(A)	
read A	

(f) (4 points) Write down an interleaving of the operations across $T1$ and $T2$ that is serially equivalent, but impossible with strict two-phase locking. Explain what makes the interleaving impossible with strict two-phase locking.

Solution: One possible solution:

$T1$	$T2$
write E	
read D	
read B	
read A	
	write A
	write B
write C	
	write C
	read B
	read D

Since all conflicting pairs are in the order of $T1, T2$, it is serially equivalent with $T1$ followed by $T2$. But $T2$'s first operation need to acquire write_lock(A), which is blocked by $T1$'s read A operation (since it holds read_lock(A)).

(g) (4 points) Is there a *non-serial* interleaving of operations across $T1$ and $T2$, that could result from using *timestamped ordering*, and is equivalent in effect to a serial execution of $T1$ followed by $T2$ (without any of these transaction getting aborted)? If not, explain why. If yes, provide an example interleaving and indicate the relevant state maintained by the objects for timestamped ordering. You can use the example format like "write X (X.committedTS = 1, X.RTS = [1,2], X.TW=[2])" to indicate the state maintained by the object (i.e. timestamps for reads and tentative writes) after an operation has been executed.

Solution: Yes it's possible. We use the ordering from the previous part:
$T1$ write E: (E.committedTS = 0, E.RTS = [], E.TW = [1])
$T1$ read D: (D.committedTS = 0, D.RTS = [1], D.TW = [])
$T1$ read B: (B.committedTS = 0, B.RTS = [1], B.TW = [])
$T1$ read A: (A.committedTS = 0, A.RTS = [1], A.TW = [])
$T2$ write A: (A.committedTS = 0, A.RTS = [1], A.TW = [2])
$T2$ write B: (B.committedTS = 0, B.RTS = [1], B.TW = [2])
$T1$ write C: (C.committedTS = 1, C.RTS = [], C.TW = [])
$T2$ write C: (C.committedTS = 1, C.RTS = [], C.TW = [2])

$T2$ read B: (B.committedTS = 0, B.RTS = [1], B.TW = [2])
$T2$ read D: (D.committedTS = 0, D.RTS = [1,2], D.TW = [])

3. **Two-Phase Commit**

Figure 1 shows a system of three servers processing a distributed transaction. Server 1 is the coordinator and interacts with the client. The network delay between the client and the coordinator, and among the three servers is indicated in the figure.

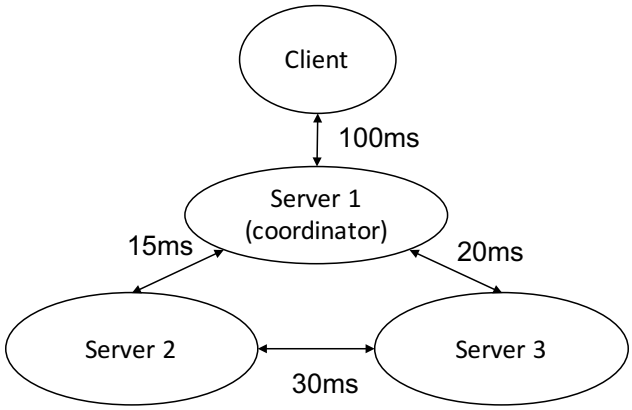

Figure 1: Figure for question 3

Any local processing at a server or self-messages take negligible time. The client issues a COMMIT request for its transaction at time t=0s. Assuming no messages are lost, no server crashes, and no server wishes to abort the transaction. Answer the following questions:

(a) (3 points) When will each of the three servers locally commit the transaction?

Solution: Server 1 receives the request from client at 100ms.
Server 1 will commit at 140ms when it receives replies from both server 2 and server 3.
Server 2 will commit at 155ms when it receives server 1's COMMIT message.
Server 3 will commit at 160ms when it receives server 1's COMMIT message.

(b) (2 points) What is the earliest time at which the coordinator can safely send a message to the client stating that the transaction will be successfully committed?

Solution: The coordinator can send the message to the client at 140ms, since all servers have replied at this time.