Homework 4: Solution

Solution 1:

- The remote reference module is responsible for translating between local and remote object references, while the binder handles mappings from textual names to remote object references.
- Binder is a separate shared service while the remote reference module is not. There is a remote reference module at each of the processes.
- When a new remote object is seen by the remote reference module, it creates a remote object reference and adds it to the table. On the other hand, the binder never creates a mapping by itself – it is the responsibility of the servers to register their remote objects by name in the binder.

Solution 2:

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.

Solution 3:

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.

Solution 4:

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, j
		write(i, 55)	
		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 i and j. i is accessed by T before U and 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 i=55, j= 66.

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

Solution 5:

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.

Solution 6:

Scheme:

- When transaction T blocks on waiting for transaction U, add edge T → 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 → U
- V requests write a_i. Add V → 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 → T.

Modification:
Store both direct and indirect edges. In our example, when transaction T blocks on waiting for transaction U add edge T → U then, when V starts waiting add V → U and V → T

Solution 7:

In the decentralized 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^2.
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^2 instead of 3N.

Solution 8:

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 → U → T.

Solution 9:

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.

Solution 10:

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.