Distributed Systems: Concepts and Design

Chapter 17 Exercise Solutions

17.1 In a decentralized variant of the two-phase commit protocol the participants communicate directly with one another instead of indirectly via the coordinator. In Phase 1, the coordinator sends its vote to all the participants. In Phase 2, if the coordinator's vote is *No*, the participants just abort the transaction; if it is *Yes*, each participant sends its vote to the coordinator and the other participants, each of which decides on the outcome according to the vote and carries it out. Calculate the number of messages and the number of rounds it takes. What are its advantages or disadvantages in comparison with the centralized variant?

17.1 Ans.

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.

17.2 A three-phase commit protocol has the following parts:

Phase 1: is the same as for two-phase commit.

Phase 2: the coordinator collects the votes and makes a decision; if it is *No*, it *aborts* and informs participants that voted *Yes*; if the decision is *Yes*, it sends a *preCommit* request to all the participants. participants that voted *Yes* wait for a *preCommit* or *doAbort* request. They acknowledge *preCommit* requests and carry out *doAbort* requests.

Phase 3: the coordinator collects the acknowledgments. When all are received, it *Commits* and sends *doCommit* to the participants. participants wait for a *doCommit* request. When it arrives they *Commit*.

Explain how this protocol avoids delay to participants during their 'uncertain' period due to the failure of the coordinator or other participants. Assume that communication does not fail.

17.2 Ans.

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.

17.3 Explain how the two-phase commit protocol for nested transactions ensures that if the top-level transaction commits, all the right descendents are committed or aborted.

17.3 Ans.

Whenever a nested transaction commits, it reports its status and the status of its descendants to its parent. Therefore when a transaction enters the committed state, it has a correct list of its committed descendants. Therefore when the top-level transaction starts the two-phase commit protocol, its list of committed descendants is correct. It checks the descendants and makes sure they can still commit or must abort. There may be nodes that ran unsuccessful descendants which are not included in the two-phase commit protocol. These will discover the outcome by querying the top-level transaction.

17.4 Give an example of the interleavings of two transactions that is serially equivalent at each server but is not serially equivalent globally.

17.4 Ans.

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 $T \rightarrow U \rightarrow T$.

17.5 The *getDecision* procedure defined in Figure 17.4 is provided only by coordinators. Define a new version of *getDecision* to be provided by participants for use by other participants that need to obtain a decision when the coordinator is unavailable.

Assume that any active participant can make a *getDecision* request to any other active participant. Does this solve the problem of delay during the 'uncertain' period? Explain your answer.

At what point in the two-phase commit protocol would the coordinator inform the participants of the other participants' identities (to enable this communication)?

17.5 Ans.

The signature for the new version is:

getDecision (trans) -> Yes/ No/ Uncertain

The worker replies as follows:

If it has already received the *doCommit* or *doAbort* from the coordinator or received the result via another worker, then reply *Yes* or *No;*

if it has not yet voted, reply *No* (the workers can abort because a decision cannot yet have been reached);

if it is uncertain, reply *uncertain.*

This does not solve the problem of delay during the 'uncertain' period. If all of the currently active workers are uncertain, they will remain uncertain.

The coordinator can inform the workers of the other workers' identities when it sends out the *canCommit* request.

17.6 Extend the definition of two-phase locking to apply to distributed transactions. Explain how this is ensured by distributed transactions using strict two-phase locking locally.

17.6 Ans.

Two-phase locking in a distributed transaction requires that it cannot acquire a lock at any server after it has released a lock at any server.

A client transaction will not request *commit* (or *abort*) (at the coordinator) until after it has made all its requests and had replies from the various servers involved, by which time all the locks will have been acquired. After that, the coordinator sends on the *commit* or *abort* to the other servers which release the locks. Thus all locks are acquired first and then they are all released, which is two-phase locking

---

17.7 Assuming that strict two-phase locking is in use, describe how the actions of the two-phase commit protocol relate to the concurrency control actions of each individual server. How does distributed deadlock detection fit in?

17.7 Ans.

Each individual server sets locks on its own data items according to the requests it receives, making transactions wait when others hold locks.

When the coordinator in the two-phase commit protocol sends the *doCommit* or *doAbort* request for a particular transaction, each individual server (including the coordinator) first carries out the commit or abort action and then releases all the local locks held by that transaction.

(Workers in the uncertain state may hold locks for a very long time if the coordinator fails)

Only transactions that are waiting on locks can be involved in deadlock cycles. When a transaction is waiting on a lock it has not yet reached the client request to commit, so a transaction in a deadlock cycle cannot be involved in the two-phase commit protocol.

When a transaction is aborted to break a cycle, the coordinator is informed by the deadlock detector. The coordinator then sends the *doAbort* to the workers.

17.8 A server uses timestamp ordering for local concurrency control. What changes must be made to adapt it for use with distributed transactions? Under what conditions could it be argued that the two-phase commit protocol is redundant with timestamp ordering?

17.8 Ans.

Timestamps for local concurrency control are just local counters. But for distributed transactions, timestamps at different servers must have an agreed global ordering. For example, they can be generated as (local timestamp, server-id) to make them different. The local timestamps must be roughly synchronized between servers.

With timestamp ordering, a transaction may be aborted early at one of the servers by the read or write rule, in which case the *abort* result is returned to the client. If a server crashes before the client has done all its actions at that server, the client will realise that the transaction has failed. In both of these cases the client should the send an *abortTransaction* to the coordinator.

When the client request to *commit* arrives, the servers should all be able to commit, provided they have not crashed after their last operation in the transaction.

The two-phase commit protocol can be considered redundant under the conditions that (i) servers are assumed to make their changes persistent before replying to the client after each successful action and (ii) the client does not attempt to commit transactions that have failed.

17.9 Consider distributed optimistic concurrency control in which each server performs local backward validation sequentially (that is, with only one transaction in the validate and update phase at one time), in relation to your answer to Exercise 17.4. Describe the possible outcomes when the two transactions attempt to commit. What difference does it make if the servers use parallel validation?

17.9 Ans.

At server *X*, *T* precedes *U*. At server *Y*, *U* precedes *T*. These are not serially equivalent because there are Read/ Write conflicts.

T starts validation at server *X* and passes, but is not yet committed. It requests validation at server *Y*. If *U* has not yet started validation, *Y* can validate *T*. Then *U* validates after *T* (at both). Similarly for *T* after *U*.

T starts validation at server *X* and passes, but is not yet committed. It requests validation at server *Y*. If *U* has started validation, *T* will be blocked. When *U* requests validation at *X*, it will be blocked too. So there is a deadlock.

If parallel validation is used, *T* and *U* can be validated (in different orders) at the two servers, which is wrong.

17.10 A centralized global deadlock detector holds the union of local wait-for graphs. Give an example to explain how a phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure.

17.10 Ans.

A centralized global deadlock detector holds the union of local wait-for graphs. Give an example to explain how a phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure.

Suppose that at servers *X*, *Y* and *Z* we have:

X	*Y*	*Z*	*global detector*
$T \rightarrow U$	$U \rightarrow V$	$V \rightarrow T$	
	U aborts		$T \rightarrow U U \rightarrow V V \rightarrow T$
$T \rightarrow U$	-	$V \rightarrow T$	
-	-	$V \rightarrow T$	$V \rightarrow T$

when *U* aborts, *Y* knows first, then *X* finds out, eventually the global detector finds out, but by then it may be too late (it will have detected a deadlock).

17.11 Consider the edge chasing algorithm (without priorities). Give examples to show that it could detect phantom deadlocks.

17.11 Ans.

Transaction *U*, *V* and *W* perform operations on a data item at each of the servers *X*, *Y* and *Z* in the following order:

U gets data item at *Y*

V gets data item at *X* and then blocks at *Y*.

W gets data item at *Z*

U blocks at *Y*.

W blocks at *X*.

V aborts at *Y*

The table below shows three servers *X*, *Y* and *Z*, the transactions they coordinate, the holders and requesters of their data items and the corresponding wait-for relationships before *V* aborts:

X (coordinator of: *V*)	*Y* (coordinator of: *U*)	*Z* (coordinator of: *W*)
held by: *V* requested by: *W*	held by: *U* requested by: *V*	held by: *W* requested by: *U*
$W \rightarrow V$ (blocked at *Y*)	$V \rightarrow U$ (blocked at *Z*)	$U \rightarrow W$ (blocked at *X*)

Now consider the probes sent out by the three servers:

At server *X*: $W \rightarrow V$ (which is blocked at *Y*); probe $<W \rightarrow V>$ sent to *Y*; then

At *Y*: probe $<W \rightarrow V>$ received; observes $V \rightarrow U$ (blocked at *Z*), so adds to probe to get $<W \rightarrow V \rightarrow U>$ and sends it to *Z*.

When *V* aborts at *Y*; *Y* tells *X* - the coordinator of *V*, but *V* has not visited *Z*, so *Z* will not be told about the fact that *V* has aborted!

At *Z*: probe $<W \rightarrow V \rightarrow U>$ is received; *Z* observes $U \rightarrow W$ and notes the cycle $W \rightarrow V \rightarrow U \rightarrow W$ and as a result detects phantom deadlock. It picks a victim, (presumably not *V*).

17.12 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*, *U* and *V* are defined as follows:

T: x= Read (i); Write(j, 44);

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

V: *Write(k, 77);Write(k, 88);*

Describe the information written to the log file on behalf of these three transactions if strict two-phase locking is in use and *U* acquires a_i and a_j before *T*. Describe how the recovery manager would use this information to recover the effects of *T*, *U* and *V* when the server is replaced after a crash. What is the significance of the order of the commit entries in the log file?

17.12 Ans.

As transaction *U* acquires a_j first, it commits first and the entries of transaction *T* follow those of *U*. For simplicity we show the entries of transaction *V* after those of transaction *T*.

P_0: ...	P_1: Data: i 55	P_2: Data: j 66	P_3: Trans: *U* prepared <i, P_1> <j, P_2> P_0	P_4: Trans: *U* commit P_3;
P_5: Data: j 44	P_6: Trans: *T* prepared <j, P_5> P_4	P_7: Trans: *T* commit P_6;	P_8: Data: k 88	P_9: Trans: *V* prepared <k,P_8> P_7
P_{10}: Trans: *V* commit P_9				

The diagram is similar to Figure 14.9. It shows the information placed at positions P_0, P_1, ...P_{10} in the log file.

On recovery, the recovery manager sets default values in the data items a_1..a_n. It then starts at the end of the log file (at position P_{10}). It sees *V* has committed, finds P_9 and *V*'s intentions list <*k*,P_8> and restores a_k=88. It then goes back to P_7 *(T commit)*, back to P_6 for *T*'s intentions list <*j*,P_5> and restores a_j=44. It then goes back to P_4 (*U commit*), back to P_3 for *U*'s intentions list <*i*,P_1><*j*, P_2>. It ignores the entry for a_j because it has already been recovered, but it gets a_i=55. The values of the other data items are found earlier in the log file or in a checkpoint.

The order of the commit entries in the log file reflect the order in which transactions committed. More recent transactions come after earlier ones. Recovery starts from the end, taking the effects of the most recent transactions first.

17.13 The appending of an entry to the log file is atomic, but append operations from different transactions may be interleaved. How does this affect the answer to Exercise 17.12?

17.13 Ans.

As there are no conflicts between the operations of transaction *V* and those of *T* and *U*, the log entries due to transaction *V* could be interleaved with those due to transactions *T* and *U*. In contrast, the entries due to *T* and *U* cannot be interleaved because the locks on a_i and a_j ensure that *U* precedes *T*.

17.14 The transactions *T*, *U* and *V* of Exercise 17.12 use strict two-phase locking and their requests are interleaved as follows:

T	*U*	*V*
x = Read(i);		
		Write(k, 77);
	Write(i, 55)	
Write(j, 44)		
		Write(k,88)
	Write(j, 66)	

Assuming that the recovery manager appends the data entry corresponding to each *Write* operation to the log file immediately instead of waiting until the end of the transaction, describe the information written to the log file on behalf of the transactions *T*, *U* and *V*. Does early writing affect the correctness of the recovery procedure? What are the advantages and disadvantages of early writing?

17.14 Ans.

As *T* acquires a read lock on a_i, *U*'s *Write(i,55)* waits until *T* has committed and released the lock:

P_0: ...	P_1:Data: k 77	P_2:Data: j 44	P_3:Trans: *T* prepared <j, P_2> P_0	P_4:Data: k 88
P_5: Trans: *V* prepared <k,P_4> P_3	P_6: Trans: *T* commit P_5	P_7: Trans: *V* commit P_6	P_8:Data: i 55;	P_9: Data: j 66
P_{10}: Trans: *U* prepared <i,P_8> <j,P_9> P_7	P11: Trans: *U* commit P_{10}			

We have shown a possible interleaving of *V*'s *Write(k,88)* and *prepared* entries between *T*'s *prepared* and *commit* entries. Early writing does not affect the correctness of the recovery procedure because the *commit* entries reflect the order in which transactions were committed.

Disadvantages of early writing: a transaction may abort after entries have been written, due to deadlock. Also there can be duplicate entries (like *k*=77 and *k*=88) if the same data item is written twice by the same transaction.

Advantages of early writing: commitment of transactions is faster.

17.15 The transactions *T* and *U* are run with timestamp ordering concurrency control. Describe the information written to the log file on behalf of *T* and *U*, allowing for the fact that *U* has a later timestamp than *T* and must wait to commit after *T*. Why is it essential that the commit entries in the log file should be ordered by timestamps? Describe the effect of recovery if the server crashes (i) between the two *Commits* and (ii) after both of them.

T	*U*
x= Read(i);	
	Write(i, 55);
	Write(j, 66);
Write(j, 44);	
	Commit
Commit	

What are the advantages and disadvantages of early writing with timestamp ordering?

17.15 Ans.

The timestamps of *T* and *U* are put in the recovery file. Call them $t(T)$ and $t(U)$, where $t(T)<t(U)$.

P_0: ...	P_1: Data: i 55	P_2: Data: j 66	P_3: Data: j 44	P_4:Trans: t(U) prepared <i,P_1> <j,P_2> P_0
P_5:Trans: t(U) waiting to commit P_4	P_6:Trans: t(T) prepared <j,P_4> P_5	P_7: Trans: t(T) commit P_6	P_8: Trans: t(U) commit P_7	

The entry at P_5 shows that *U* has committed, but must be ordered after *T*. If the transaction *T* aborts or the server fails before *T* commits, this entry indicates that *U* has committed.

It essential that the commit entries in the log file should be ordered by timestamps because recovery works through the log file backwards. The effects of later transactions must be over write the effects of earlier ones.

The effect of recovery:

(i) if the server crashes between the two commits, we lose the entry at P_8, but we have P_6: *Trans t(U) waiting to commit*. Transaction *U* can be committed as the transaction it waits for has either committed, or if it has not yet committed, it will be aborted.

(ii)if the server crashes after both the commits, both will be recovered.

The advantage of early writing with timestamp ordering is that commitment is quicker. Transactions can always commit if they get that far (clients don't abort them). There do not appear to be any important disadvantages.

17.16 The transactions *T* and *U* in Exercise 17.15 are run with optimistic concurrency control using backward validation and restarting any transactions that fail. Describe the information written to the log file on their behalf. Why is it essential that the commit entries in the log file should be ordered by transaction numbers? How are the write sets of committed transactions represented in the log file?

17.16 Ans.

P_0: ...	P_1: Data: i 55	P_2: Data: j 66	P_3: Trans: T_U prepared <i,P_1> <j,P_2> P_0	P_4: Trans: T_U commit P_3
P_5: Data: j 44	P_6:Trans T_T prepared <j,P_4> P_5	P_7: Trans T_T commit P_6		

Transaction numbers rather than transaction identifiers (or timestamps) go in the recovery file after a transaction has passed its validation. *U* passes validation because it has no *read* operations. Suppose *U* is given the transaction number T_U. Transaction *T* fails validation because its read set {*i*} overlaps with *U*'s Write set {i,j}. *T* is restarted after *U*. It passes validation with transaction number T_T. where $T_T > T_U$

It essential that the commit entries in the log file should be ordered by transaction number because transaction numbers reflect the order in which transactions are committed at the server. Recovery takes transactions from newest to oldest by reading the log file backwards.

Write sets are represented in the log file in the prepared entries.

17.17 Suppose that the coordinator of a transaction crashes after it has recorded the intentions list entry but before it has recorded the participant list or sent out the *canCommit?* requests. Describe how the participants resolve the situation. What will the coordinator do when it recovers? Would it be any better to record the participant list before the intentions list entry?

17.17 Ans.

As the coordinator is the only server to receive the *closeTransaction* request from the client, the workers will not know the transaction has ended, but they can time out and unilaterally decide to abort the transaction (see pages 521-3). They are allowed to do this because they have not yet voted. When the coordinator recovers it also aborts the transaction.

An apparent advantage of recording a worker list earlier (before the coordinator fails), is that it could be used to notify the workers when a coordinator recovers, with a view to avoiding the need for timeouts in workers. Unfortunately workers cannot avoid the need for timeouts because the coordinator may not recover for a very long time.
