The University of Melbourne
School of Computing and Information Systems
COMP90020 Distributed Algorithms

Tutorial Week 12: Distributed Transactions.

Solutions

51. Since the transaction outcome is known to each participant before the participants make the decisive move into COMMITTED or ABORTED state, independent recovery is possible even if the coordinator fails. However consistent and independent recovery can only be guaranteed in the event of at most one failure. It follows that in 3PC the uncertain period lasts only from when the participant votes yes until to receives a precommit message, after which if the a single participant or coordinator crashes they can recover independently. In 2PC the uncertainty extends from when the participant votes yes but has not yet been told the outcome. It can no longer abort and must block on the response i.e. hold locks on the objects. Note that additional approaches exist that allow for cooperative termination via maintaing a list of all participants at each participant in the case where the coordinator fails (they can gossip the result).

52.

We will be consistent with more formal transaction literature in this example and slightly vary our notation. Where $q_i(x), q \in w, r$ denotes an operation by transaction i on object x. We won't specifcally care about the actual value written or read since it isn't relevant here.

Let s be a global history with local histories $s_1, ..., s_n$ involving a set of T transactions such that each $s_i, 1 \leq i \leq n$, is conflict serializable.

s is globally conflict serializable iff there exists a total order $<$ on T that is consistent with the local serialization orders of the transactions i.e.

$$(\forall t, t' \in T, t \neq t')t < t' \implies$$
$$(\forall s_i, 1 \leq i \leq n, t, t' \in trans(s_i))(\exists s_i', s_i' serial, s_i \approx_c s_i')t <_{s_i'} t'$$

Where $\approx_c$ represents conflict serial equivalence.

We present two local histories:

$$s_1 : r_1(x)w_2(x)$$
$$s_2 : r_2(y)w_1(x)$$

This leads to locally serial histories $t_1 <_{s_1} t_2, t_2 <_{s_2} t_1$.

Assume it is globally serial, so there must exist a total ordering $<$ on T which implies $(t_1 <_{s_1} t_2 \wedge t_1 <_{s_2} t_2) \vee (t_2 <_{s_1} t_1 \wedge t_2 <_{s_2} t_1)$. Yet, $t_1 <_{s_1} t_2 \wedge t_2 <_{s_2} t_1$, therefore there cannot be a total ordering and as a result s is not globally conflict serializable.

53. 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 released, which satisfies 2PL.

54. Consider a cycle that exists $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_1$.

If both t_1 and t_3 initiate an edge chasing algorithm at the same time they will come to the same cyclic graphs yet they both might decide to abort. A possible solution is to totally order the Txs (unique integer identifiers). So that once a cycle has been determined, the min(cycle) can be aborted. In this way, two separate Tx both initiating this algorithm at the same time will come to the same tx to be aborted. Could we then extend this further and not relay messages where the message is also not forwarded in the case where the identifier is less than your own (ring algorithm?). Instead putting your own if you are waiting? (reduce message complexity)

55.

(a) Every transaction (excluding the root) must respond yes exactly once and likewise have been asked to prepare exactly once. It follows that the $\sum YES = \sum PREPARE$. Likewise for Commit/Ack. Therefore, it should be the case that every non-root node has received exactly 2 messages and responded to those 2 messages (success). We can take the number of messages as $4(m-1) = 4(2^n - 2)$. The number of non-root nodes multiplied by 4. For the forced log writes, there are a total of 2^{n-1} leaf nodes, $2^{n-1} - 2$ non-root, non-leaf (internal) and 1 root. All the leaf nodes write twice. All the non-leaf, non-root nodes write four times. The root writes twice. Therefore $2(2^{n-1}) + 4(2^{n-1} - 2) + 2 = 3(2^n - 2)$

(b) This case is identical to above, force-write commit entries are swapped with rollback entries.