==========

DISTRIBUTED COMPUTING (CS 380D) : Fall 2013

Project 2 Writeup

Vineet Keshari (EID vk3226) Slip days used : [on this project: 2, total: 6]
Vijay Talluru (EID vt4225) Slip days used : [on this project: 2, total: 4]

==========

Section 0: Problems 3 and 5 implemented

The bank application is implemented in BankState.java. Each replica maintains a copy of a BankState to which it makes reads and updates.

The logic for keeping only one leader active at a given time is implemented in Leader.java. Each leader maintains a Map<ProcessId, Long> leaderTimeouts that contains the next time value in milliseconds to consider leader with key dead. The size of this map is either 0 (active) or 1 (passive, pinging) at any given time.

Section 1: Paxos algorithm modified for Read-Only commands as described in the paper.

The suggestion in the paper is as follows:
If leader receives a RO command, it:

1. Spawn a scout to check if ballot number is current
2. On getting an Adopted message from the Scout, it decides and sends the RO command to all replicas (skips getting agreement from a majority of acceptors).

We implement these changes as follows:
For part 1:

On receiving a command, if the leader is active:

If message is RO, spawn a scout.

else, spawn a commander as before.

If the leader isn't active then it is already waiting for an adopted message from a Scout to become active again, which will inform it if its ballot is current.

For part 2:

On receiving an adopted message, spawn a commander as usual.

In commander,

If message is RO, send the decision directly to replicas.

else, send P2a messages and wait for majority of acceptors as before, and then send the decisions.

We will now show that this suggestion, as given in the paper and implemented as above, is incorrect.

Section 2: Paxos algorithm modified for Read-Only commands as above is not correct.

Consider the following setup. There are two leaders L1 and L2. There are two replicas R1 and R2. There are three acceptors A1, A2 and A3. There can be a maximum of 1 failure in this system.

Two commands C1 and C2 are requested. C1 is a Read-Only (RO) command and C2 is an Update command. The following situation occurs:

- Replica R1 gets the commands in the order C1(RO), C2
- Replica R2 gets the commands in the order C2, C1(RO)

Let's follow the replicas, acceptors and leaders through the execution steps.

1. In the beginning:
    a. L1 is active with its ballot_number=b1
    b. L2 is waiting on L1 by pinging it with ballot_number=b0 < b1
2. R1 receives <request,C1> (RO)
3. R1 proposes <propose, s1,C1> to L1 where s1 is a slot number.
4. L1 on receiving the proposal <s1,C1>, since it is a read-only command, spawns a Scout for b1 (as per the modified Paxos algorithm)
5. A1, A2 and A3 adopt b1 and respond with their *accepted* set to the scout. The *accepted* set for each of them is currently empty.
6. L1 spawns commander <b1, s1, C1>
7. The commander will send <decision,s1,C1> to the replicas without consulting the acceptors.
8. R1 receives the decision <decision,s1,C1>, executes C1, sets slot_num to s2.
9. **BOOM! L1 crashes (which also takes down the commander with it). So, R2 never receives the decision <decision,s1,C1>.**
10. As mentioned before, L2 is the other leader, It is in passive mode. When the ping to L1 times out, L2 will run a scout. The scout will return a preempted message with ballot number b1. This will make L2 set its ballot_num to b2 > b1.
11. Now, L2 will spawn a scout with b2 > b1.
12. A1, A2 and A3 will adopt b2 and the scout will return an *adopted* message to the leader.
13. L2 becomes active.
14. At this stage, R2 receives C2 and proposes <s1,C2> . C2 is an update command.
15. L2 receives the proposal and spawns a commander <b2 s1 C2>.
16. A1, A2 and A3 will accept <b2,s1,C2> i.e They add it to their *accepted* list. Note that the *accepted* lists were empty so far.
17. The commander will send <decision,s1,C2> to R1 and R2.
18. R1 receives the <decision,s1,c2>. s1 is less than the current slot number of R1. R1 will overwrite the mapping <s1,C1> with <s1,C2> but never executes it.

19. R2 receives <decision,s1,C2> . It executes C2 and sets the slot number to s2.

20. Now, R2 will receive the <request,C1> (RO). It proposes <propose, s2, C1 >.
21. L2 will receive this proposal. Since it is a read only message, on receiving this proposal, L2 will spawn a scout with its current ballot_num b2.
22. A1, A2 and A3 will respond with b2 and their accepted list, containing <b2,s1,C2>, to the scout and the scout will send an adopted message to L2.
23. L2, will add <s1,C2> to its proposals.
24. L2 will spawn a commander <b2,s2,C1>. The commander will send the decision <decision,s2,C1> to R2.
25. R2 will receive the above decision and it will add it to its own *decisions.*
26. R2 *performs* C1 and sets slot_num to s3.

27. R1 will do the same and sets its slot number to s3.

At this point in the execution, the replicas R1 and R2 are no more in identical state.
R1 has executed the following commands in the order shown
<s1, C1> (RO), <s2, C1> (RO)
and R2 has executed the following commands in the order shown
<s1, C2>, <s2, C1> (RO)

Any client receiving the response for C1 from R1 will be reading a stale data (as C2 is not performed on R1). It is clear that the following invariant of Paxos is violated:

Invariant r1: There are no two different commands decided for the same slot:

$$\forall s,\ \rho_1, \rho_2, p_1, p_2\ :\ <s, p_1>\ \in\ \rho_1.decisions\ \wedge\ <s, p_2>\ \in\ \rho_2.decisions\ \Rightarrow\ p_1 = p_2.$$

Step 19 onwards, in the above execution, slot s1 contains commands C1 and C2 in replicas R1 and R2 respectively.

Hence we have shown an execution using the modified paxos algorithm which results in incorrect execution.

Section 3(a): Whats wrong with the modified Paxos algorithm
We make the following observation:

The above execution leads to an inconsistent state because the acceptors did not have an entry for the RO command C1 in their *accepted* set in step 16. Suppose the following happened as per the older algorithm without modifications:

- In step 7, the acceptors would have been consulted for <b1, s1, C1> by the commander, and they would have added <b1, s1, C1> to their *accepted* sets.
- Then, the scout would have returned this *accepted* set of pvalues to L2 in step 12, and L2 would have over-written its entry for s1 in its *proposals* map from <s1, C2> to <s1, C1>
- R1 and R2 would therefore have never received and executed <s1, C2>. Instead, they would receive <s1, C1>, and consistency would be maintained. We would therefore not violate invariant r1 on step 19, or thereafter.

We use the above observation to propose the following solution to this problem.

Section 3(b): Solution to making the modified Paxos algorithm work

When a leader spawns a commander to execute an RO command, instead of avoiding sending P2a messages to the acceptors altogether, the commander:

- First sends a modified P2aRO message to the acceptors.
- It then sends the decision to all replicas without waiting for any responses.

On receiving a P2aRO message, an acceptor:

- Adds pvalue <ballot_num, slot_num, command> from P2aRO to its *accepted* set, as it would do for a p2a message.
- However, it doesn't send a P2b response to the commander.

The P2aRO message therefore acts as a notification from the commander to update its *accepted* set of pvalues. As discussed in section 3(a) above, if the *accepted* set is updated we will not run into the problem described in section 2.

This modification of commander and acceptor behaviors, and P2aRO messages, have been implemented in the submitted code.