Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.824 Distributed System Engineering: Spring 2015

Exam I Solutions

I Lab 2

Ben Bungler wants to build a variant of Lab 2 that uses two backups instead of one.

Ben modifies his view server so that it puts two backups into each view (if machines are available). Ben's view server designates them backup1 and backup2. If only one backup machine is available, the view server makes it backup1. If the primary fails, the view server promotes backup1 to be the new primary. If backup1 dies or is promoted, the view server promotes backup2 (if any) to be backup1.

Ben is worried that the primary's network connection to the backups might be slow or unreliable. He reasons that it's enough to have a copy of each operation at just one of the two backups, so he modifies his primary code to forward each operation at first to just backup1; if the RPC fails, then only in that case does the primary forward to backup2; if that RPC fails, the primary tries backup1 again; until the RPC to one of them succeeds. Ben knows that this design can't tolerate two server failures, but his main focus is allowing the primary to continue quickly in the face of slow or unreliable connectivity to the backups.

1. [6 points]: Describe a scenario in which clients of Ben's design will see a serious consistency problem.

Answer: The client sends put(x,99) to the primary; the primary forwards the put only to backup2; the primary fails and backup1 becomes the new primary; now a get(x) won't return 99.

Ben realizes his design has a flaw. Here's his plan for fixing it. If the primary dies, then before taking over as primary, backup1 will send its entire key/value database to backup2.

2. [6 points]: Explain why Ben's "fixed" design still doesn't work.

Answer: The previous answer works here too.

II Lab 3

Here's a copy of the Paxos pseudo-code from Lab 3:

```
   --- Paxos Proposer ---

 1 proposer(v):
 2    while not decided:
 2     choose n, unique and higher than any n seen so far
 3     send prepare(n) to all servers including self
 4     if prepare_ok(n, na, va) from majority:
 5       v' = va with highest na; choose own v otherwise
 6       send accept(n, v') to all
 7       if accept_ok(n) from majority:
 8         send decided(v') to all


   --- Paxos Acceptor ---

 9 acceptor state on each node (persistent):
10  np     --- highest prepare seen
11  na, va --- highest accept seen

12 acceptor's prepare(n) handler:
13  if n > np
14    np = n
15    reply prepare_ok(n, na, va)
16   else
17     reply prepare_reject


18 acceptor's accept(n, v) handler:
19  if n >= np
20    np = n
21    na = n
22    va = v
23    reply accept_ok(n)
24   else
25     reply accept_reject
```

3. [6 points]: Why must a proposer receive prepare_ok from a majority before moving to the accept phase?

Answer: The majority ensures that a new proposer is guaranteed to see any value that might already have been agreed on (i.e. accepted by a majority).

Lab 3 solutions must exchange Done() values to allow peers to forget instances that no peer will need again. Unfortunately, if one peer is offline for an extended period of time, the Done() mechanism will prevent any peers from forgetting instances. Ben Bitdiddle thinks he can do better than this.

Ben wants his servers to be able to discard certain Put operations regardless of whether all servers are advancing Done(). He figures that, if a server applies one Put operation and then a second Put for the same key, the server can safely forget the first Put. Any peers asking for the old Put operation might as well skip it (treat it as a no-op), since they'll see the later Put. Ben plans to modify his implementation so that a server discards a superseded Put by replacing it with a No-Op in his Paxos set of remembered instances. A No-Op takes less memory to store than a Put.

4. [6 points]: Is Ben's plan for forgetting Puts correct? Explain your answer.

Answer: Suppose we have a three-replica system with servers S1, S2, and S3. Suppose key x starts out non-existent. Client C1 sends a put(x,1) to S1, which uses Paxos to add put(x,1) to the log in slot 20. S2 doesn't immediately see the accept message for slot 20. C1 then sends a get(x) to S2, which proposes the get(x) for slot 21; agreement is reached for the get(x) in slot 21 but S2 doesn't know it yet. Server S1 receives a put(x,2), uses Paxos to add the put(x,2) to the log in slot 22. Once servers S1 and S3 have processed the put(x,2), they replace the put(x,1) in slot 20 with a no-op. Now S2 completes agreement for slot 20 and sees the no-op, completes agreement for slot 21 and sees that slot 21 holds C1's get(x), and (because there is no put(x,1) before the get(x)) returns a "no such value" error to C1. C1 knows that can't be right because it had previously completed a put(x,1).

Ben is having trouble answering the above question so he thinks about discarding Get operations instead. In his implementation a No-Op consumes less memory than a Get. Ben plans to modify his servers to replace each Get instance with a No-Op after it has been agreed to and executed.

5. [6 points]: Is Ben's plan for forgetting Gets correct? Explain your answer.

Answer: It's correct. Unlike Puts, Gets don't modify any client-visible state. Gets need to be in the log in order to ensure that they see state after all previously applied updates, so that (for example) a client never sees a Get reply that doesn't reflect its own Puts, no matter which server it talks to. It's true that the Gets in the log may populate server duplicate RPC tables, but this is not necessary: it's OK for a server to handle a re-sent Get by treating it as an entirely new operation.

In Paxos as implemented in Lab 3, servers lose state when they crash and therefore can't re-join the Paxos group. Ben Bitdiddle thinks he can fix this by having recovering peers initialize themselves from a live peer by cloning the live peer's state. (Note: the recovering peer doesn't clone the live peer's identity, just its Paxos state). This turns out not to work.

6. [6 points]: Why doesn't this work?

Answer: The recovering peer might have been part of a bare majority that agreed on a particular value. If it copies state from a live peer that wasn't part of that majority, that would allow Paxos to agree to a different value for the same instance.

III Remus

Recall the Remus paper by Cully *et al.* Suppose there are two clients talking to a service. Client 1 sends requests sequentially, waiting for a response from the service for each request before sending the next request. Client 2 sends many requests concurrently; that is, it sends a batch of requests to the server before waiting for any responses. The clients initially both send requests to a single non-replicated server. However, we decide to make the service more fault tolerant by replicating it with Remus. For these questions, there are no failures, the network adds no latency to communication, and the client CPU is infinitely fast.

7. [6 points]: Which client will experience the bigger slowdown in throughput when we upgrade to Remus? Briefly explain your answer.

Answer: Client 1. Before Remus, both clients could keep the service 100% busy. With Remus, each client only receives replies to requests once per epoch. Consequently, Client 1 achieves a throughput of only 1 request per epoch, while Client 2 achieves throughput of n requests per epoch, where n is the number of requests it can send concurrently.

Now calculate the average throughput that client 1 and client 2 can achieve. Throughput is the number of requests acknowledged by the server per second. Client 2 keeps five requests concurrently outstanding. An epoch is 1 second long. The system pauses for 0.1 seconds to get the snapshot of its state and then takes another 0.1 seconds to transfer it to the backup. It takes 0.5 seconds for the backup to copy all of the changes to its buffer and ack back to the primary. Assume all other network and processing latencies are zero.

8. [6 points]:

Your answer for client 1: **Answer:** 1 request / 1.1 second

Your answer for client 2: **Answer:** 5 requests / 1.1 second

IV Raft

Consider the Raft replication system described in *In Search of an Understandable Consensus Algorithm* by Ongaro and Ousterhout. Suppose you have a five-server Raft system. Here's the state of the servers' logs. The notation T.N means the Nth log entry from term T. The servers are about to choose a new leader.

```
S1: 3.1 4.1
S2: 3.1 3.2
S3: 3.1 5.1
S4: 3.1 5.1 5.2
S5: 3.1
```

9. [6 points]: Could any replica have already executed the operation in log entry 5.1? If yes, explain how this could have happened. If no, explain why it could not have happened.

Answer: No. An operation cannot execute if it is not in a majority of logs.

10. [6 points]: Could operation 3.2 be committed in the future? (Assume that the client does **not** re-transmit the operation.) If yes, how could that happen? If no, why not?

Answer: No. 3.2 can only be committed if S2 becomes the next leader (since any other leader will cause S2 to delete 3.2). S2 can't become a leader because there is no majority in which S2 has the most up-to-date log.

11. [6 points]: Could operation 4.1 be committed in the future? (Assume that the client does **not** re-transmit the operation.) If yes, how could that happen? If no, why not?

Answer: Yes. S1 can become the next leader with a majority from S1, S2, and S5.

12. [6 points]: A classmate points out that read-only operations could be executed much faster if the Raft leader alone executed them and sent the result back to the client, without sending such operations to the followers. After all, read-only operations don't modify the state, so executing them on the followers is a no-op. Explain why this idea would lead to incorrect behavior.

Answer: That would allow a leader in a network partition to continue to reply to client requests even after a new leader is elected. The old leader might then serve stale data, since it would not know about the latest write operations.

Imagine that you are using Raft to replicate a file server, so that the state being replicated is the file system stored on disk, and executing a client operation involves updating the file system on disk. Suppose a client operation has been committed to the Raft log in position 20. The leader has replied to the client, and the client never re-sends the request.

13. [6 points]: A follower crashes and re-starts. Could Raft give the operation in position 20 to the follower's state machine for execution twice, once before the crash and once after? That is, could the "apply log[lastApplied] to state machine" in the paper's Figure 2 Rules for Servers ever happen twice for the same log entry on the same replica? If yes, explain how that could happen. If no, explain why it couldn't happen.

Answer: Yes. Figure 2 says lastApplied is in volatile state, so the re-starting server won't know which operations it has already applied to its on-disk state.

V Treadmarks

Consider the Treadmarks distributed shared memory system described in the paper by Keleher *et al.*

14. [6 points]: Describe a situation in which Treadmarks would move some memory content from processor Px to processor Py even though the program on Py never uses that memory.

Answer: Suppose variables v1 and v2 are on the same page. Px writes v1, acquires a lock, writes v2, and releases the lock. Py acquires the lock, which will cause Py to mark the page invalid. Py then reads v2. Py will now fetch all write differences for the page from Px, which will include v1, even though Px isn't reading v1.

Now suppose you are using Treadmarks for a parallel computation on three processors, P1, P2, and P3. Each runs the code indicated here:

```
P1:
  mu1.Lock()
  for(i = 0; i < 100; i++)
    x[i] = random()
  mu1.Unlock()

P2:
  mu2.Lock()
  for(j = 0; j < 100; j++)
    y[j] = random()
  mu2.Unlock()

P3:
  mu1.Lock()
  for(k = 0; k < 100; k++)
    x[k] = x[k] + 10
  mu1.Unlock()
```

`x` and `y` are arrays in shared memory managed by Treadmarks. `x` and `y` are in the same page of memory. `i`, `j`, and `k` are local to the processor that uses each (they are not shared). `mu1` and `mu2` are locks managed by Treadmarks. `random()` returns a random number.

15. [6 points]: Suppose P1 acquires `mu1` first, and then P3 acquires `mu1`. Treadmarks will move the `x` array to P3. Explain how Treadmarks is able to move much less memory to P3 than IVY would (IVY is the system described in *Memory Coherence in Shared Virtual Memory Systems* by Li and Hudak).

Answer: Treadmarks will only copy the x array to P3. Treadmarks can do this because P1 and P2 can hold *different* versions of the same page, and because P3 knows (because of the lock) that it only needs updates from P1. IVY keeps just one copy of the page (bouncing between P1 and P2), so it sends the whole page to P3, including both x and y.

VI 6.824

16. [5 points]: Which were the most and least interesting papers?

Most "interesting" votes: Raft, Paxos, Treadmarks, Harp, FDS.

Most "not interesting" votes: Ficus, IVY, MapReduce, Harp.

17. [5 points]: How can we improve 6.824?

The most frequent answers:

- Lecture should answer each paper question.
- Hand out a working Paxos after Lab 3A due date.
- Document the lab tests.
- Hold Go tutorial sessions.
- Assign fewer papers.
- Provide lab debugging tools.

End of Exam I Solutions