IV Raft

Refer to Ongaro and Ousterhout's *In Search of an Understandable Consensus Algorithm (Extended Version)*.

Consider the following optimization for Raft: instead of calling `persist()` whenever the log changes in the AppendEntries RPC handler, the implementation calls `persist()` in a background thread every 100ms to reduce the persistence overhead.

> **5. [5 points]:** Describe a sequence of events that would lead to the implementation committing two different entries at the same index.

Answer: Consider a three-node cluster.

1. Server 1 as a leader sends AppendEntries RPC containing an entry X to Server 2 and 3.
2. A network issue drops the RPC message to Server 3.
3. Server 2 receives X from Server 1, appends X to its log, and replies positively.
4. On receiving the reply from Server 2, Server 1 commits the entry by counting replicas.
5. A network issue puts Server 1 on its own partition.
6. Server 2 crashes before writing X to the stable storage.
7. Server 2 recovers and together with Server 3 they elect Server 3 as a leader in a new term.
8. Following the successful path of AppendEntires RPC, Server 2 and 3 commits entry Y at the same index as X.

Alyssa hears about a new cool technology: persistent memory (e.g., Intel Optane). Persistent memory behaves like a DRAM memory but is nonvolatile: it retains its content in the event of a crash—that is, if a server reboots, the content of persistent memory contains the values from before the crash. Alyssa equips each server in the Raft cluster with persistent memory.

> **6. [5 points]:** Alyssa stores all the variables listed in Figure 2 in a `Raft struct`. Alyssa modifies her Raft library to store this `Raft struct` in persistent memory at a well-known address. She removes the persistor and the calls to it. When a server reboots, it sets the `Raft struct` pointer to the well-known address, and initializes the volatile state to 0.
>
> Alyssa notices that this implementation can result in incorrect behavior. Explain why.

Answer: Updates to persistent state aren't atomic: On a RequestVote RPC, for example, the code may update `currentTerm`, crash before updating votedFor, reboot, and now incorrectly vote for a new leader in that term.

V Raft and ZooKeeper

Refer to Ongaro and Ousterhout's *In Search of an Understandable Consensus Algorithm (Extended Version)* and *ZooKeeper: Wait-free coordination for Internet-scale systems* by Hunt, Konar, Junqueira, and Reed.

Ben wants to simplify his Raft implementation, so he decides to use ZooKeeper for leader election instead of implementing it in Raft. He removes the `RequestVote` RPC and the `votedFor` state from his implementation. In his new version of Raft, when a server wants to become leader in term `newTerm`, instead of becoming a Candidate and sending `RequestVoteRPC`s, the server attempts to `create()` a regular znode with path `/raftLeader/{newTerm}` in ZooKeeper. If the `create` fails (which happens when the znode already exists), the Raft server does not become leader. Otherwise, if the create succeeds, the Raft server becomes leader in `newTerm`. The rest of the Raft implementation (e.g. what happens after a server becomes a leader) is unchanged from Ben's initially correct version of Raft.

7. [5 points]: Explain how this modification to Raft can result in incorrect behavior.

Answer: The problem with removing the `RequestVote` RPC is that nothing else in Raft enforces the election restriction. In this modified version, a leader could be elected that is not as up to date as its followers.
Here's an example execution with 3 servers A, B, C. Initially, C is partitioned away from A and B. Server A becomes leader in term 1 by creating the znode and then commits operation $[op_x]$ at index 1 by replicating it to server B. Then, server C is reconnected to A and B. It becomes a leader in term 2 by creating the znode `/raftLeader/2`. Now that it is leader, C puts a different client operation $[op_y]$ at index 1 in its log. C overwrites A and B's log (because its entry is in a higher term) to eventually commit op_y at index 1. This is inconsistent with op_x being committed at index 1 by server A.

VI Linearizability

These questions concern the material from Lecture 9, Consistency and Linearizability.

A service's state consists of just one value, a string. There are two operations: append to the string, and read the entire string. In history diagrams,

```
|--Axyz--|
```

means a client issued a request to append "xyz" to the string, and

```
|--Rabxyz--|
```

means a client issued a read, and the response was "abxyz".

8. [4 points]: The service's string starts out empty. The only requests are those shown in the following history. Is this history linearizable?

```
|----Ax----| |----Ab----|
      |---Ay---|
                 |---Ryxb---|
```

Answer: Yes. The numbers below mark linearization points that are consistent with the results:

```
|----Ax---2| |-3---Ab----|
      |-1--Ay---|
                 |-4--Ryxb---|
```

9. [4 points]: Again, the service's string starts out empty, and there are no other requests other than the ones shown in the following history. Which choices of XXX result in a linearizable history? For each possibility, circle either YES or NO.

```
  |------Axy------|
|----Aab----| |----Aq----|
        |----RXXX----|
```

A.	xyab	YES	NO
B.	axbqy	YES	NO
C.	abq	YES	NO
D.	xyq	YES	NO

Answer: **A** and **C** are YES, **B** and **D** are NO. **A** is possible if the serial order is Axy, Aab, Rxyab, Aq. **B** is not possible because any serial order of the operations must execute them one at a time, so while `abxy` and `xyab` are possible, `axby` (for example) is not. **C** corresponds to the serial order Aab, Aq, Rabq, Axy. **D** is not possible because Aab finishes before Aq starts, and thus Aab must occur before Aq in any legal serial order, but `xyq` does not have `ab` before the `q`.