

\*\*\* GFS

Consider the paper "The Google File System" by Ghemawat et al.

Suppose that, at the moment, nobody is writing to the GFS file named "info". Two clients open "info" and read it from start to finish at the same time. Both clients' cached meta-data information about file "info" is correct and up-to-date. Are the two clients guaranteed to see the same content? If yes, explain how GFS maintains this guarantee; if no, describe an example scenario in which the clients could see different content.

Answer: No, the two clients may see different content. Two chunkservers with the same version of the same chunk may store different content for that chunk. This can arise if a client previously issued a record append to the file, and the primary asked the secondaries to execute the append, but one of the secondaries didn't receive the primary's request. The primary doesn't do anything to recover from this; it just returns an error to the writing client. Thus if the two clients read from different chunkservers, they may see different bytes.

\*\*\* Raft (1)

Ben Bitdiddle is working on Lab 2C. He sees that Figure 2 in the Raft paper requires that each peer remember currentTerm as persistent state. He thinks that storing currentTerm persistently is not necessary. Instead, Ben modifies his Raft implementation so that when a peer restarts, it first loads its log from persistent storage, and then initializes currentTerm from the Term stored in the last log entry. If the peer is starting for the very first time, Ben's code initializes currentTerm to zero.

Ben is making a mistake. Describe a specific sequence of events in which Ben's change would lead to different peers committing different commands at the same index.

Answer: Peer P1's last log entry has term 10. P1 receives a VoteRequest for term 11 from peer P2, and answers "yes". Then P1 crashes, and restarts. P1 initializes currentTerm from the term in its last log entry, which is 10. Now P1 receives a VoteRequest from peer P3 for term 11. P1 will vote for P3 for term 11 even though it previously voted for P2.

\*\*\* Raft (2)

Bob starts with a correct Raft implementation that follows the paper's Figure 2. However, he changes the processing of AppendEntries RPCs so that, instead of looking for conflicting entries, his code simply overwrites the local log with the received entries. That is, in the receiver implementation for AppendEntries in Figure 2, he effectively replaces step 3 with "3. Delete entries after prevLogIndex from the log." In Go, this would look like:

```
rf.log = rf.log[:args.PrevLogIndex+1]
rf.log = append(rf.log, args.Entries...)
```

Bob finds that because of this change, his Raft peers sometimes commit different commands at the same log index. Describe a specific sequence of events in which Bob's change causes different peers to commit different commands at the same log index.

Answer:

- (0) Assume three peers, 1 2 and 3.
- (1) Server 1 is leader on term 1
- (2) Server 1 writes "A" at index 1 on term 1.
- (3) Server 1 sends AppendEntries RPC to Server 2 with "A", but it is delayed.
- (4) Server 1 writes "B" at index 2 on term 1.

- (5) Server 2 acknowledges ["A", "B"] in its log
- (6) Server 1 commits/applies both "A" and "B"
- (7) The delayed AppendEntries arrives at Server 2, and 2 updates its log to ["A"]
- (8) Server 3, which only got the first entry ["A"], requests vote on term 2
- (9) Server 2 grants the vote since their logs are identical.
- (10) Server 3 writes "C" at index 2 on term 2.
- (11) Servers 2 and 3 commit/apply this, but it differs from what Server 2 committed.

### \*\*\* ZooKeeper

Section 4.3 (typo: should be 4.4) of the ZooKeeper paper says that, in some circumstances, read operations may return stale values. Consider the Simple Locks without Herd Effect example in Section 2.4. The `getChildren()` in step 2 is a read operation, and thus may return out-of-date results. Suppose client C1 holds the lock, client C2 wishes to acquire it, and client C2 has just called `getChildren()` in step 2. Could the fact that `getChildren()` can return stale results cause C2 to not see C1's "lock-" file, and decide in step 3 that C2 holds the lock? It turns out this cannot happen. Explain why not.

Answer: ZooKeeper does promise that each of a client's operations executes at a particular point in the overall write stream, and that each of a client's operations executes at a point at least as recent as the previous operation. This means that the `getChildren()` will read from state that is at least as up to date as the client's preceding `create()`. The lock holder must have executed its `create()` even earlier. So the `getChildren()` is guaranteed to see the lock file created by the lock holder.

### \*\*\* CRAQ

Refer to the paper "Object Storage on CRAQ" by Terrace and Freedman.

Item 4 in Section 2.3 says that, if a client read request arrives and the latest version is dirty, the node should ask the tail for the latest committed version number. Suppose, instead, that the node replied with that dirty version (and did not send a version query to the tail). This change would cause reads to reflect the most recent write that the node is aware of. Describe a specific sequence of events in which this change would cause a violation of the paper's goal of strong consistency.

(Note that this question is not the same as the lecture question posted on the web site; this exam question asks about returning the most recent dirty version, whereas the lecture question asked about returning the most recent clean version.)

Answer: Suppose the chain consists of servers S1, S2, S3. The value for X starts out as 1. Client C1 has issued a write that sets X to 2, and the write has reached S1, but not S2 or S3. Client C2 reads X from S1 and sees value 2. After C2's read completes, client C2 reads X again, this time from S3, and sees value 1. Since the order of written values was 1, then 2, and the reads observed 2, then 1, there is no way to fit the writes and the two reads into an order that obeys real time. So the result is not linearizable.

Note that two clients seeing different values if they read at the same time is not a violation of linearizability. Both reads are concurrent with the write, so one read can be ordered before the write, and the other after the write. It is only if the reads are *not* concurrent with each other, and the second one yields an older value than the first one, that there is a violation.

### \*\*\* Frangipani

Consider the paper "Frangipani: A Scalable Distributed File System" by Thekkath et al.

Aviva and Chetty work in adjacent cubicles at Yoyodyne Enterprises.