Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.5840 Distributed System Engineering: Spring 2023

Exam I

Write your name on this cover sheet. If you tear out any sheets, please write your name on them. You have 80 minutes to complete this quiz.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, write down any assumptions you make. Write neatly. In order to receive full credit you must answer each question as precisely as possible.

You may use class notes, papers, and lab material. You may read them on your laptop, but you are not allowed to use any network. For example, you may not look at web sites, use ChatGPT, or communicate with anyone.

I (5)	II (10)	III (5)	IV (10)	V (5)	VI (10)	VII (12)	VIII (3)	Total (60)

Name:

Gradescope E-Mail Address:

Grade histogram for Exam 1

$$\begin{aligned} \text{max} &= 60 \\ \text{median} &= 51 \\ \mu &= 50.35 \\ \sigma &= 5.86 \end{aligned}$$

I MapReduce

You have a small MapReduce computation whose Map phase produces the following six intermediate key/value data items:

```
a/1
c/2
b/3
c/4
b/5
b/6
```

R (the number of reduce tasks) is two, and the two reduce tasks are numbered zero and one. There are two single-core worker machines available to run reduces. It takes a worker machine one second to process each intermediate key/value pair, so if there were just one worker the Reduce phase would last six seconds.

Section 3.1 of Dean and Ghemawat's MapReduce paper discusses hashing the keys of intermediate data. The hash function takes an intermediate key as argument, and returns a number. Your job is to pick the hash function that will cause the reduce phase to take the shortest amount of wall-clock time for the above intermediate data. You do not have to worry about the behavior of the hash function in other situations. Circle the best answer.

1. [5 points]:

A. a $\rightarrow$ 0, b $\rightarrow$ 0, c$\rightarrow$0

B. a $\rightarrow$ 0, b $\rightarrow$ 0, c$\rightarrow$1

C. a $\rightarrow$ 0, b $\rightarrow$ 1, c$\rightarrow$0

D. a $\rightarrow$ 0, b $\rightarrow$ 1, c$\rightarrow$1

E. All of the above choices result in the same amount of time.

Answer: Choice **C** (a $\rightarrow$ 0, b $\rightarrow$ 1, c$\rightarrow$0). The hash function determines which of the two worker machines executes the reduce function for each key. With choice **C**, both workers process three intermediate items, so the reduce phase finishes in three seconds. With the other choices, one of the workers processes four or more items, so the reduce phase would take four or more seconds.

II GFS

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

2. [5 points]: Explain briefly how GFS ensures that there is at most one active primary for a given chunk.

Answer: The master hands out a lease to a new primary only after the current lease has expired.

A GFS file starts out containing the single byte "a". The file's single chunk is replicated on three chunkservers. Client C1 opens the file and issues a write with offset zero and data "b" (i.e. C1 asks to replace the file's content with "b"), and waits for the reply from GFS. After seeing the reply to the first write, C1 issues a write with offset zero and data "c", and waits for the reply. At about the same time, client C2 runs three iterations of a loop that opens the file and reads it; C2 waits for the reply to each operation before starting the next. GFS replies with success to every one of C1's and C2's requests. There are no other client requests.

3. [5 points]: What sequences of read results could C2 see? Circle either YES or NO for each sequence below.

A. Sequence: a, a, a YES NO
B. Sequence: a, b, a YES NO
C. Sequence: a, c, a YES NO
D. Sequence: c, b, a YES NO
E. Sequence: c, b, b YES NO

Answer: Sequences A, B, E are YES; the others are NO. Sequence A can occur if the primary executes all three of C2's reads before C1's first write. Sequence B can occur if the primary has applied C1's first write to its own copy but before the primary has forwarded the write to the backups; then C2 sends its first read to a backup and gets "a", sends its second read to the primary and gets "b", and sends its third read to a backup and gets "a". Sequence C cannot occur: if any replica has seen the write of "c", that means the write of "b" has finished (and was successful, which means all replicas executed it), so no replica could still contain "a". Sequence D cannot occur for the same reason. Sequence E can occur if the primary has applied C1's second write to its own copy but before the primary has forwarded the write to the backups; then C2 sends its first read to the primary and gets "c", sends its second read to a backup and gets "b", and sends its third read to a backup and gets "b".

III VMware FT

The Design of a Practical System for Fault-Tolerant Virtual Machines, by Scales et al., says in Section 2.2 and Figure 2 that the Output Rule must be enforced for network packets sent to clients.

Consider the following scenario with a primary, backup, one client, and an output instruction i telling the primary's network card to send a TCP packet to the client. FT delays sending this packet until it receives an acknowledgment from the backup for i. The primary crashes either before or after receiving the backup's acknowledgment and thus either sends or does not send the delayed packet. When the backup takes over it will first replay its log, which includes an entry for i. In replay mode the backup will not generate output, including for instruction i. Then, after replaying the log, the backup becomes the new primary.

4. [5 points]: Even though FT doesn't send the TCP packet for i in replay mode, it is possible for a client to receive the TCP packet twice. Explain how.

Answer: The network may drop the TCP acknowledgment from the client, which will cause the TCP software on the new primary to timeout and resend the TCP packet. The client has received the packet now twice: once from the old primary and once from the new primary.

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 `RequestVoteRPCs`, 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`.

VII ZooKeeper

Refer to *ZooKeeper: Wait-free coordination for Internet-scale systems* by Hunt, Konar, Junqueira, and Reed.

Alyssa has a ZooKeeper installation with three ZooKeeper replica servers (a leader and two followers). Alyssa runs the following two programs, M3 and M4, each on a separate client machine, starting them at about the same time.

```
M3:
  create("/m3", "c3po")
  if exists("/m4") == NO:
    print "three"
  print "done"

M4:
  create("/m4", "r2d2")
  if exists("/m3") == NO:
    print "four"
  print "done"
```

The `create()` and `exists()` calls invoke the ZooKeeper client library. The znodes involved, `/m3` and `/m4`, start out not existing, and both `create()` calls succeed. The `create()` calls ask for normal znodes, not ephemeral. `exists()` has one of three results: it returns `YES` if the znode exists, `NO` if the znode does not exist, and it causes the program to immediately exit if there was an error (e.g., if ZooKeeper says it has terminated the session). The requests are all synchronous. There are no other client requests.

10. [3 points]: Can Alyssa see *both* M3 print “three” and M4 print “four”?

Answer: No. The ZooKeeper leader orders the two create()s in one order or the other, and all followers see them in that order. If M3 prints ”three”, then create(m3) must have been ordered first. That means all followers will see create(m3) and then create(m4). M4’s follower won’t process M4’s exists(m3) until after it has seen the ZooKeeper leader send M4’s preceding create(m4) (by the FIFO client ordering rule), and thus after it has seen M3’s create(m3), so M4’s exists will return YES, and it won’t print “four”.

11. [3 points]: Is it possible for both programs to print just “done” alone?

Answer: Yes. If ZooKeeper completely processes both create()s before either client sends its exists(), then both exists() will return YES, so neither “three” nor “four” will be printed.

Alyssa is experimenting with coordinator election using ZooKeeper. When she starts, no znodes exist on her ZooKeeper service (other than the root). Alyssa starts both these programs at about the same time on two different client machines:

Machine M1 executes the following code in single session:

```
if create("/coordinator", ephemeral=true):
  print "M1 won"
  create("/m1", "xyzzy")
  if exists("/m2") == YES:
    print "M1 sees M2"
  while true:
    act as coordinator...
else:
  exit
```

Machine M2 executes the following code in a single session:

```
if create("/coordinator", ephemeral=true):
  print "M2 won"
  create("/m2", "xyzzy")
  if exists("/m1") == YES:
    print "M2 sees M1"
  while true:
    act as coordinator...
else:
  exit
```

There is no client activity other than these two programs.

12. [3 points]: Could Alyssa see *both* “M1 won” and “M2 won”?

Answer: Yes. M1 could win the race to create /coordinator, print “M1 won”, and then fail, causing ZooKeeper to delete the ephemeral /coordinator. Then M2 could create /coordinator and print “M2 won”.

13. [3 points]: Could Alyssa see *both* “M1 sees M2” and “M2 sees M1”?

Answer: No. Suppose M1 prints “M1 sees M2”. This can only happen if M2 had successfully created /coordinator first and then created /m2, but then ZooKeeper had terminated its session and deleted the ephemeral /coordinator, allowing M1 to successfully create it. If ZooKeeper terminated M2’s session before ZooKeeper executed M2’s exists(/m1), then M2 could not have seen exists() return YES, and could not have printed “M2 sees M1”. If ZooKeeper terminated M2’s session *after* ZooKeeper executed M2’s exists(/m1), then (since M1 at that point had not yet even created /coordinator) M2’s exists(/m1) must have returned NO, so M2 could not have printed “M2 sees M1”. Thus, if M1 prints “M1 sees M2”, M2 cannot print “M2 sees M1”.

VIII 6.5840

14. [1 points]: Which lectures/papers should we definitely keep for future years?

- MapReduce
- RPC, Threads, and Go
- GFS
- VMware FT
- Raft
- Q+A on Lab 2A/2B
- Linearizability testing
- ZooKeeper
- Chain Replication

Answer: Keep Raft, MapReduce, and GFS.

15. [1 points]: Which lectures/papers should we omit?

- MapReduce
- RPC, Threads, and Go
- GFS
- VMware FT
- Raft
- Q+A on Lab 2A/2B
- Linearizability testing
- ZooKeeper
- Chain Replication

Answer: Get rid of Chain Replication.

16. [1 points]: What can we do to improve the course?

Answer: Better debugging tools for the labs.

End of Exam I