Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.5840 Distributed System Engineering: Spring 2024

Exam I

Please write your name on the bottom of each page. 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.

Name:

Gradescope E-Mail Address:

Grade histogram for Exam 1

$$\begin{aligned} \text{max} &= 98.25 \\ \text{median} &= 81.25 \\ \mu &= 80.32 \\ \sigma &= 9.72 \end{aligned}$$

I MapReduce

Have a look at Figure 3(a) in the paper *MapReduce: Simplified Data Processing on Large Clusters* by Dean and Ghemawat. The three graphs on the left show the rate of data movement over time for a MapReduce job that sorts a terabyte of data: the rate at which Maps read their input, the rate at which intermediate data is shuffled, and the rate at which Reduces write their output. For these questions you should assume that only this MapReduce job is using the servers and network, and that there are no failures. Many of the numbers below are derived from looking at the graphs, and are thus approximate; your reading of the graphs may be somewhat different from our's; you should circle the answer that is closest to what you think is correct.

1. [6 points]: Roughly when is the first time at which the sort application's Reduce() function is called? Circle the best answer.

- 0 seconds
- 50 seconds
- 150 seconds
- 300 seconds

Answer: The best answer is 150 seconds. No Reduce function can be called until every Map function has finished; the top graph suggests that the Maps stop running around 150 seconds, and the paper text mentions 200 seconds.

2. [7 points]: Roughly how long does it take a single application Reduce function to sort its share of the data (just the sort, not including either the shuffle or the writing of the output)? Circle the best answer.

- 10 seconds
- 75 seconds
- 200 seconds
- 250 seconds
- 650 seconds

Answer: This question is broken: the application Reduce function does not sort the data. MapReduce's reduce task framework does the sort, and (for this application) the application Reduce function just returns its argument.

Name: _____

3. [6 points]: Why are there two bumps in the Shuffle graph? That is, why does the Shuffle graph go up and then down from time 20 to 200, remain at zero for 100 seconds, and then go up and then down from time 300 to 600? Circle the best answer.

- There are more Map tasks (M = 15,000) than there are machines.
- There are more Reduce tasks (R = 4000) than there are machines.
- There are more Map tasks than there are Reduce tasks.
- The aggregate network throughput is smaller than the aggregate disk throughput.
- The Map tasks consume more CPU time than the Reduce tasks.

Answer: The best answer is the second one (more Reduce tasks than machines). Intermediate data can only be moved from Map machines to Reduce machines for Reduce tasks that have been allocated to machines. There are only 1800 machines, so at first only 1800 of the 4000 Reduce tasks are assigned to machines, so only about 1800/4000ths of the shuffles can happen at first. That's the first bump. The second bump starts once the first set of Reduce tasks finishes, moving intermediate data to the machines that will run the remaining Reduces.

4. [7 points]: Why does the shuffle begin a long time before the Map phase has finished? Circle the best answer.

- There are more Map tasks (M = 15,000) than there are machines.
- There are more Reduce tasks (R = 4000) than there are machines.
- There are more Map tasks than there are Reduce tasks.
- The aggregate network throughput is smaller than the aggregate disk throughput.
- The Map tasks consume more CPU time than the Reduce tasks.

Answer: The best answer is the first one (more Map tasks than machines). Shuffles can start as soon as Map functions finish. The system runs 1800 Maps at a time; the first of these finishes a long time before the last of the 15,000 Maps finishes at time 200.

Name: _____

II Linearizability

These questions concern the material from Lecture 4, Consistency and Linearizability.

You have a service whose state is a single string, and that exposes two RPC operations to clients: one operation appends the RPC argument to the state, and the other RPC operation returns the current state. The timelines below indicate the start time, end time, argument string, and reply string for each client operation. Ax indicates an append operation with argument x, and Ry indicates a read operation to which the server replied y. The vertical bars indicate the start and end times of each operation (the times at which the client sends the request, and receives the reply). The service's state string starts out empty at the beginning of each history.

For example,

```
C1: |---Ax---|
C2:     |---Ay---|
C3:   |--Ryx--|
```

means that client C1 sent an append RPC with "x" as the argument, C2 sent an append RPC with "y" as the argument, and C3 read the state and received the reply "yx".

Name: _____

Consider this history, in which the reply string sent to C4 has been omitted:

```
C1: |---Ax---|
C2:       |---Ay---|
C3:                 |---Az---|
C4:                        |--R?--|
```

5. [6 points]: Which values could C4's read yield that are consistent with linearizability? Circle all of the correct answers.

- xzy
- yxz
- yzx
- xy
- xz
- yx
- zy

Answer: xzy, yxz, xy, and yx. The result C4 receives can't start with z (since the Az starts after the Ax finishes); if both x and z appear, x must come first; and it must include both x and y (since Ax and Ay both finish before the C4's read starts).

Name: _____

Now look at this history:

```
C1: |-------Ax-------|
C2:     |---Ay---|
C3:   |--Ry--|
C4:             |----R?----|
```

6. [7 points]: Which values could C4's read yield that are consistent with linearizability? Circle all of the correct answers.

- y
- x
- yx
- xy

Answer: y and yx. The fact that C3 read y, and that C3's read finished before C4's read started, means that C4's result must include y, and, if it includes x, the x must come after y.

Name: _____

III GFS and Raft

After reading the GFS paper (*The Google File System* by Ghemawat *et al.*) and the Raft paper (Ongaro and Ousterhout's *In Search of an Understandable Consensus Algorithm (Extended Version)*), Ben replaces the GFS master with a new coordinator that uses Raft. The Raft-based coordinator provides the same functions as before but replicates the log of operations using 3 Raft peers. All other parts of GFS stay the same.

Which of the following statements are true? (Circle all that apply)

7. [6 points]:

A. The coordinator can continue operation in the presence of network partitions without any additional monitoring infrastructure, if one partition with peers is able to achieve a majority.

B. The coordinator can continue operation correctly even if one of the 3 peers has failed (and there are no other failures).

C. None of the above are true

Answer: Both A and B are true; these are properties of Raft.

Name: _____

Ben also considers using Raft for chunk replication. He runs many Raft clusters and has the GFS master assign chunks to a specific Raft cluster (i.e., each chunk is assigned to one Raft cluster, consisting of a leader and two followers). GFS clients submit write and append operations for a chunk to the leader of the Raft cluster for that chunk (i.e., Ben's design doesn't implement the separate data flow). The leader of the Raft cluster replicates write and append operation using the Raft library. All other parts of GFS (e.g., assigning leases to the leader, client caching locations of chunk servers, reading from the closest server, and so on) stay the same. (You can assume that chunk servers have enough disk space for operations to succeed.)

Which of the following statements are true? (Circle all that apply)

8. [7 points]:

A. Unlike the old design, Ben's design can achieve linearizability for chunk operations.

B. Unlike the old design, Ben's design can continue operation despite the failure of one chunk server.

C. By using Raft, Ben's design allows clients to perform more mutating chunk operations per second than the old design.

D. Raft's snapshots allow a chunk server to catch up in a few seconds if has been down for a long time (assuming the same network as in the GFS paper).

E. None of the above are true

Answer: None of the above are true. **A** is false because the client's cache that maps file names to chunk handles can yield stale results. **B** is false because the old design can continue despite one failure as well. **C** is false because Ben's scheme moves data less efficiently (via the leader, rather than the separate data flow). **D** is false because the snapshot mechanism sends the leader's entire database of chunks, which will likely take far longer than a few seconds.

Name: _____

IV Raft

Consider the Raft paper (Ongaro and Ousterhout's *In Search of an Understandable Consensus Algorithm (Extended Version)*). Ben wonders what the impact of network behavior is on Raft's performance. Ben runs a Raft-replicated server that receives many client requests. If the network delivers AppendEntries RPCs in order, Ben's Raft implementation is fast (i.e., completes many client requests per second). But, if the network delivers AppendEntries frequently out of order, Ben's Raft implementation performs badly (i.e., completes fewer client requests per second). Using the rules in Figure 2 explain why this is the case.

9. [6 points]:

Answer: This question is broken. Figure 2 implies that each AppendEntries should include all as-yet-unacknowledged log entries. So if there are two such RPCs outstanding, the one that was sent second contains a copy of all the log entries in the first. This means that, if the second RPC arrives first, it will be accepted. So it's not clear why Ben would see any different performance due to out-of-order delivery.

Name: _____

V Lab 3A-3C

Alyssa is implementing Raft as in Lab 3A-3C. She implements advancing the commitIndex at the leader (i.e., last bullet of Leaders in Fig 2) as follows:

```
func (rf *Raft) advanceCommit() {
    start := rf.commitIndex + 1
    if start < rf.log.start() { // on restart start could be 1
        start = rf.log.start()
    }
    for index := start; index <= rf.log.lastindex(); index++ {
        if rf.log.entry(index).Term != rf.currentTerm { // 5.4
            continue  // ***
        }
        n := 1 // leader always matches
        for i := 0; i < len(rf.peers); i++ {
            if i != rf.me && rf.matchIndex[i] >= index {
                n += 1
            }
        }
        if n > len(rf.peers)/2 { // a majority?
            DPrintf("%v: Commit %v\n", rf.me, index)
            rf.commitIndex = index
        }
    }
}
```

Assume that all omitted parts of Alyssa's code are correct.

Name: _____

Ben argues that the line marked with "***" could be replaced by a break statement so that the loop terminates immediately.

10. [7 points]: Explain what could go wrong if one adopted Ben's proposal; please include a specific sequence of events to illustrate your answer.

Answer: If there's a term mis-match, the leader won't be able to commit any further log entries. The paper's Figure 8e shows an example of such a scenario.

Name: _____

VI More lab 3A-3C

Alyssa is implementing Raft as in Lab 3A-3C. She implements the rule for conversion to follower in her AppendEntries RPC handler as shown below:

```
func (rf *Raft) convertToFollower(term int) {
    rf.state = Follower
    rf.votedFor = -1
    rf.currentTerm = term
    rf.persist()

}


func (rf *Raft) AppendEntries(args *AppendEntriesArgs,
                              reply *AppendEntriesReply) {
    rf.mu.Lock()
    defer rf.mu.Unlock()

    if args.Term >= rf.currentTerm {

        rf.convertToFollower(args.Term)

    }

    ...

}
```

Assume that all omitted parts of Alyssa's code are correct.

Name: _____

11. [6 points]: Describe a specific sequence of events that would cause Alyssa's implementation to break the safety guarantees provided by Raft.

Answer: The code shown can cause a peer to forget it has cast a vote for the current term. Suppose peer P1 has been elected for this term. The peers that elected it may forget that they voted for P1. Then some other peer P2 may become candidate for this term, and get votes from those forgetful peers, and become a second leader for the same term. This will lead to split brain.

VII ZooKeeper

Refer to *ZooKeeper: Wait-free coordination for Internet-scale systems* by Hunt, Konar, Junqueira, and Reed, and to the notes for Lecture 9.

The code fragments below are simplified versions of how something like GFS or MapReduce might use ZooKeeper to elect a coordinator, and for that coordinator to store state such as the assignments of GFS data to chunkservers.

Suppose server S1 executes the following code to become elected and to then store coordinator state in /A and /B. Initially, znode /coord-lock does not exist, znode /A starts out containing A0, and znode /B starts out containing B0.

```
s = openSession()
if create(s, "/coord-lock", data="S1", ephemeral=true) == true:
    setData(s, "/A", "A1", version=-1)
    setData(s, "/B", "B1", version=-1)
```

12. [7 points]: Briefly explain why, for coordinator election, it makes sense that /coord-lock should be an ephemeral znode rather than a regular znode.

Answer: If a server is elected as coordinator, and then fails, ZooKeeper automatically deletes the ephemeral /coord-lock; now another server can create that file and become coordinator.

Name: _____

S1's create() finishes and returns true to indicate success. But just after that, and before ZooKeeper has received S1's setData() requests, ZooKeeper decides that S1 has failed, and ZooKeeper terminates S1's session.

After ZooKeeper terminates S1's session, server S2 runs this to become coordinator:

```
s = openSession()
if create(s, "/coord-lock", data="S2", ephemeral=true) == true:
    setData(s, "/A", "A2", version=-1)
    setData(s, "/B", "B2", version=-1)
```

However, S1 is actually still alive, and it proceeds to send the two setData() requests, and they arrive at ZooKeeper.

Then client C1 reads /B and /A and sees B2 and A2, respectively.

Now a different client, C2, reads /B, and then reads /A. Both reads succeed.

13. [6 points]: Given the way ZooKeeper works, what can C2 observe? Circle all of the possible read results.

```
/B   /A
-------
B0   A0
B0   A1
B0   A2
B2   A0
B2   A1
```

Answer: B0 A0 and B0 A2 are the only possible results. B0 is possible because, in the absence of other constraints, ZooKeeper can yield stale data to reads. A1 is never possible because ZooKeeper terminated S1's session before ZooKeeper receive S1's setData()s, so ZooKeeper ignore those setData()s. B2 A0 is not possible since, once ZooKeeper has revealed a write to a client, the "Linearizable writes" guarantee in Section 2.3 implies that all previous writes have been applied.

Name: _____

VIII Grove

In the `ApplyReadonly` function in Figure 7, Ben decides to delete the check for `s.waitForCommitted()`. The new code is as as follows:

```
func (s *Server) ApplyReadonly(op) Result {
    s.mutex.Lock()

    if s.leaseExpiry > GetTimeRange().latest {
        e := s.epoch
        idx, res := s.stateLogger.LocalRead(op)
        s.mutex.Unlock()
        return res
    } else {
        s.mutex.Unlock()
        return ErrRetry
    }
}
```

14. [7 points]: Explain why this modification can result in non-linearizable reads.

Answer: If a Grove backup server reveals an update without waiting to ensure it has been committed, then it may reveal an uncommitted write. If the primary then fails, the backup whose database is used to recover may not have recent uncommitted writes. So the write may disappear, and other clients issuing strictly subsequent reads may not see that write. That would not be linearizable.

Name: _____

IX Distributed Transactions

MouseGPT is designing a distributed transaction system using two-phase commit and two-phase locking, as discussed in Lecture 12 and Chapter 9 of the 6.033 reading. The goal is to provide serializable results. The question arises of what should happen if a participant computer crashes while in the PREPARED state for a transaction. MouseGPT thinks that all-or-nothing atomicity would be satisfied if such a transaction were completely forgotten. So MouseGPT designs the system so that if a participant computer crashes and restarts while it is in the PREPARED state for a transaction that it's part of, the recovery software on that computer un-does any local modifications the interrupted transaction might have performed and releases its locks, and sends a network message to each other participant and to the TC to tell them to undo any changes made by the transaction and to release its locks.

15. [6 points]: Explain why MouseGPT's plan would cause the system to produce non-serializable (incorrect) results.

Answer: The TC may have decided to commit the transaction, and sent out COMMIT messages to the other participating workers, and they may have committed, and revealed committed results to other transactions. At that point, there is no way to back out of the transaction without violating serializability and atomicity.

Name: _____

X 6.5840

16. [1 points]: Which lectures/papers should we definitely keep for future years?

- MapReduce
- RPC, Threads, and Go
- Linearizability
- GFS
- Raft
- ZooKeeper
- Q+A on Lab 3A/3B
- Grove
- Transactions

Answer:

Lecture	Count
MapReduce	76
RPC, Threads, and Go	50
Linearizability	64
GFS	60
Raft	93
ZooKeeper	55
Q+A on Lab 3A/3B	27
Grove	22
Transactions	48

Name: _____

17. [1 points]: Which lectures/papers should we omit?

- MapReduce
- RPC, Threads, and Go
- Linearizability
- GFS
- Raft
- ZooKeeper
- Q+A on Lab 3A/3B
- Grove
- Transactions

Answer:

Lecture	Count
MapReduce	5
RPC, Threads, and Go	13
Linearizability	10
GFS	7
Raft	0
ZooKeeper	9
Q+A on Lab 3A/3B	33
Grove	51
Transactions	7

Name: _____

18. [1 points]: What can we do to improve the course?

Answer:

General feedback	Count
Fine	8
More OH	7
Overview/motivation of all covered papers/systems	6
More visual diagrams	4
More on real-world and modern systems	4
Publish lecture question answers	4
More exercises on papers	3
More distributed systems theory	2
Raft lecture too close to 2A deadline	2
More responsive on Piazza	1
More lecture questions	1
More collaboration	1
Extra credit for project	1
More implementation details on papers	1
More support for Harvard students	1
Discuss other concurrency features (e.g. async/await)	1
Linearizability lecture earlier	1
Curate FAQs	1
Make pre-lecture quesitons optional	1

Lab-related feedback	Count
Better debugging for labs	16
Less lab dependency	8
More lab QA/recitation	7
Stronger tests for labs	6
More tutorials/FAQ on Go	3
Clearer instruction on labs	2
Better windows support	2
Weight labs more in grading	2
Spread labs out across papers (instead of all being raft-based)	2
More specific lab difficulty labels	1
Post all labs at the start	1

End of Exam I

Name: _____