Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.824 Distributed System Engineering: Spring 2017

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.

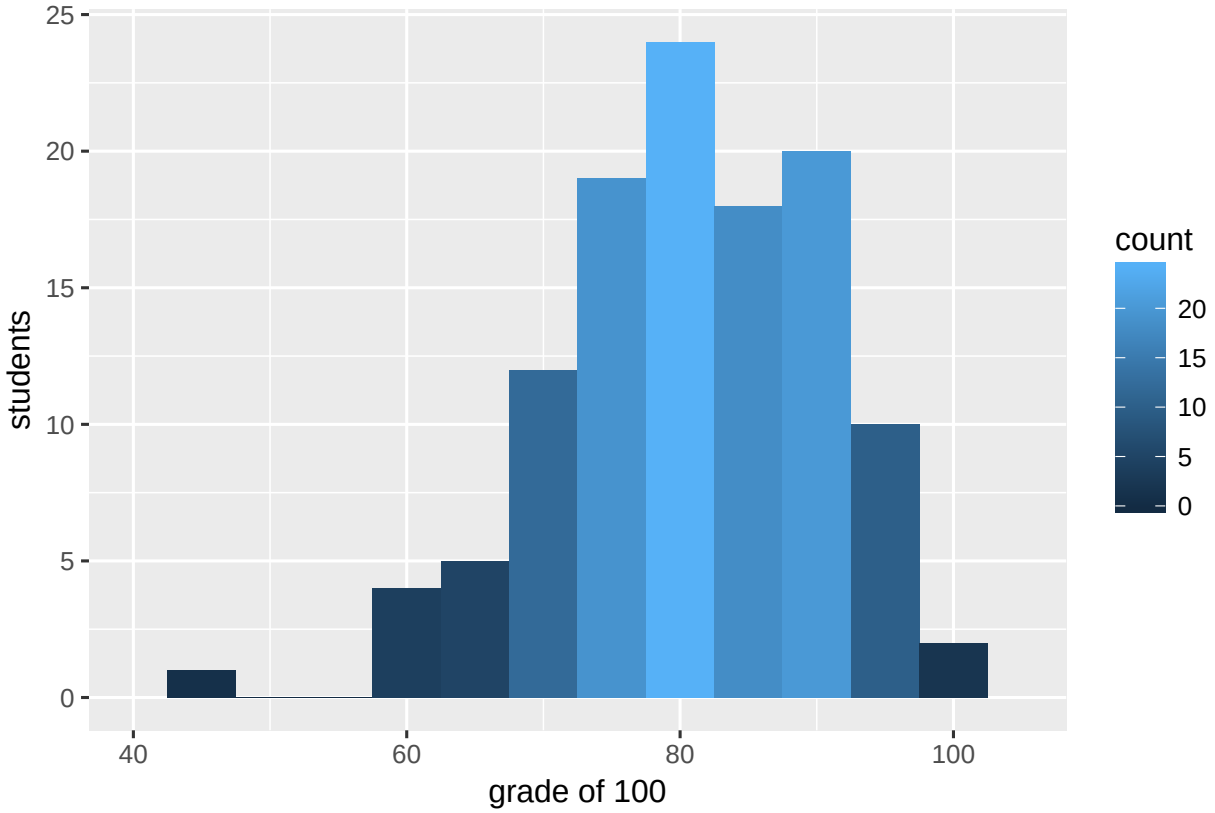
Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, write down any assumptions you make. Write neatly. In order to receive full credit you must answer each question as precisely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

I (7)	II (32)	III (21)	IV (7)	V (22)	VI (8)	VII (3)	Total (100)

Name:

Grade histogram for Exam 1



$$\begin{aligned} \text{max} &= 100 \\ \text{median} &= 81 \\ \mu &= 80.6 \\ \sigma &= 9.8 \end{aligned}$$

I Lab 1

Alyssa P. Hacker is writing a simplified version of the `schedule` function for the MapReduce lab. This function takes in an array of Tasks for a particular map or reduce phase and sends them to available workers. The function should return once all of the tasks in this phase have been completed.

```
01 func schedule(tasks []Task) {
02      var mu sync.Mutex
03      tasksRunning := 0
04      for _, task := range tasks {
05                go func(task Task) {
06                        // Count this task as running.
07                        mu.Lock()
08                        tasksRunning++
09                        mu.Unlock()
10
11                        // Get an available worker and let them run the task.
12                        worker := <-registerChan
13                        call(worker, "Worker.DoTask", task)
14
15                        // Indicate that this task has completed.
16                        mu.Lock()
17                        tasksRunning--
18                        mu.Unlock()
19
20                        // Reregister the worker as available.
21                        registerChan <- worker
22                }(task)
23      }
24      // Wait for all the tasks to be done.
25      for {
26                mu.Lock()
27                t := tasksRunning
28                mu.Unlock()
29                if t == 0 {
30                        break
31                }
32      }
33      return
34 }
```

1. [7 points]: Alyssa notices that `schedule` sometimes returns before all of the tasks have been finished, even without worker failures. Explain a sequence of events that could cause `schedule` to return too early.

Answer: The check for t == 0 at line 29 could run before any of the task goroutines start, so schedule() could return before the tasks finish.

II Short reading questions

2. [8 points]: Chunk servers in GFS (as described by Ghemawat *et al.*) store the version number of a chunk persistently on disk. How would GFS break if chunk servers stored the version number of each chunk only in memory? Briefly explain your answer.

Answer: If chunkservers don't store version numbers persistently, they will be lost after a crash. After a reboot, the GFS master won't be able to distinguish between up-to-date and stale replicas, so it wouldn't be able to do garbage collection properly.

3. [8 points]: Does Spinnaker's timeline consistency (as described in "Using Paxos to build a scalable, consistent and highly available datastore" by Rao et al.) allow a client to read a stale value? Specifically, can a client read an old value for a key after another client has successfully put a new value for that key? If yes, explain how this can happen; if no, explain why it can't happen.

Answer: Timeline consistency allows clients to read stale values (a trade-off to get better performance). Timeline consistent reads can be directed to any node in a cohort, and that node doesn't necessarily have the latest value, even if a majority of the nodes do.

In VM-FT (as described (by Scales *et al.*), when a virtual machine tries to read the time-of-day clock, the hypervisor on the primary machine obtains control and reads the real hardware time-of-day clock. The hypervisor returns the read value to the virtual machine and forwards the read value to the hypervisor on the backup machine. The hypervisor on the backup returns the value read at the primary to its virtual machine when that virtual machine reads the time-of-day clock.

4. [8 points]: Ben observes that his backup machine has a hardware time-of-day clock too, and he also sees that the time-of-day clocks on the primary and the backup are perfectly synchronized – if you read them both at the same instant, they always return exactly the same value. Ben proposes a simplified VM-FT design that avoids the communication between the primary and the backup for the time-of-day clock. When the VM on the primary wants to read the time-of-day, the hypervisor returns the value from its local clock, and doesn't forward the value to the backup. When the virtual machine on the backup wants to read the time-of-day clock, the hypervisor on the backup reads its local time-of-day clock and returns that value to its virtual machine. Even though the clocks on Ben's machines are perfectly synchronized, Ben observes that the virtual machine on the primary and the backup diverge over time. Briefly explain why Ben's design doesn't work.

Answer: The VM-FT protocol has the backup lagging behind by at least one event so by the time the backup would read the time-of-day clock, it would read a different value than the primary. This different value may cause the VM on the backup to do something different than the VM on the primary.

5. [8 points]: Ben wants to add the following scheme for duplicate request detection to ZooKeeper. His proposal is to modify Zookeeper to maintain a duplicate detection table, replicated across the Zookeeper servers. Each client stamps each of its requests with a unique ID. After Zab agrees on a request, and before a server executes it, the server indexes its table with the request's unique ID to see whether it has already executed the request. If there is no entry for that unique ID, the server executes the request and stores the result in its table under the unique ID. If the unique ID is present in the table, the server doesn't execute the request, and if it was the server that received the request from the client it sends the result stored in the table as the response to the client. This design has a serious flaw – what is it? (Briefly explain your answer.)

Answer: Ben's protocol never deletes entries from the duplicate-detection table. If a ZooKeeper server runs for a long time and serves many requests, Ben's scheme will cause it to run out of memory.

III Raft

Answer each of these questions with reference to the Raft protocol described in the paper's Figure 2 (without membership change, without log compaction, and without Section 8).

6. [7 points]: Suppose the leader waited for **all** servers (rather than just a majority) to have $matchIndex[i] >= N$ before setting commitIndex to N (at the end of Rules for Servers). What specific valuable property of Raft would this change break?

Answer: Raft would no longer be available (would no longer make progress) if there were even a single crashed or unreachable server.

7. [7 points]: Suppose the leader immediately advanced commitIndex to its last log entry without waiting for responses to AppendEntries RPCs, and regardless of the values in matchIndex. Describe a specific situation in which this change would lead to incorrect execution of the service being replicated by Raft.

Answer: If less than half of the followers received and accepted an AppendEntries RPC, then the leader could commit an entry that could be lost during a subsequent leader change. This might cause the original leader to respond positively to the client even though the request's effects might disappear. In addition the leader would not be able to correctly convert to follower, since its application state (e.g. k/v table) would reflect a non-existent operation.

8. [7 points]: There are five Raft servers. S1 is the leader in term 17. S1 and S2 become isolated in a network partition. S3, S4, and S5 choose S3 as the leader for term 18. S3, S4, and S5 commit and execute commands in term 18. S1 crashes and restarts, and S1 and S2 run many elections, so that their currentTerms are 50. S3 crashes but does not re-start, and an instant later the network partition heals, so that S1, S2, S4, and S5 can communicate. Could S1 become the next leader? How could that happen, or what prevents it from happening? Your answer should refer to specific rules in the Raft paper, and explain how the rules apply in this specific scenario.

Answer: No. S1 must get a vote from one of S4 and S5 to get an election majority, and both S4 and S5 must be aware of operations committed in term 18, since they were both part of S3's bare majority. S1's log must end with an entry no later than term 17, since it would not have been able to win an election for any later term. By the leader election restriction rule, S4 and S5 won't vote for S1 since S4 and S5's logs end with entries that have term 18.

IV Lab 2

Ben's Raft is failing the Lab 2 tests, which say that log entries are appearing on the apply channel out of order. Here is the part of his code that sends on the apply channel. Ben's implementation calls this function every time it updates the commit index. There is a single lock (Go mutex) protecting each Raft instance, and the calling code always holds the lock.

```
01. func (rf *Raft) applyToService() {
02.  for rf.commitIndex > rf.lastApplied {
03.    entry := rf.log[rf.lastApplied + 1]
04.    msg := ApplyMsg {
05.      Index: rf.lastApplied + 1,
06.      Command: entry.Command,
07.    }
08.    rf.lastApplied++
09.    go func() { rf.applyCh <- msg } ()
10.  }
11. }
```

9. [7 points]: Explain Ben's bug.

Answer: Ben is sending to rf.applyCh in a goroutine. If there are multiple such goroutines running (which can happen when multiple goroutines are spawned in that loop in a single call to applyToService), nothing prevents them from completing the sends out of the intended order (there's no explicit synchronization between the goroutines to ensure that happens).

V Concurrency Control

Suppose you have a storage system that provides serializable transactions, and that can abort transactions.

The database holds three objects, x, y, and z, with these initial values:

```
x = 1
y = 2
z = 0
```

Only two transactions execute, and they start at about the same time. This is what they do:

```
T1:
  x=10
  y=20

T2:
  temp = x + y
  z = temp
```

10. [8 points]: Which final values are possible (after each transaction has either aborted or finished committing and applying all updates)? Mark each of the following as "yes" or "no".

a. **Yes / No** x=1 y=2 z=0 **Answer:** yes

b. **Yes / No** x=1 y=2 z=30 **Answer:** no

c. **Yes / No** x=1 y=20 z=21 **Answer:** no

d. **Yes / No** x=10 y=20 z=3 **Answer:** yes

e. **Yes / No** x=10 y=20 z=30 **Answer:** yes

11. [7 points]: Suppose T2 has just executed its first line, but has not tried to commit. If the transaction system uses pessimistic two-phase locking, can the first line of T2 see x=10 and y=2? How, or why not?

Answer: No. If x is 10, but y still has its old value, then T1 must have started committing but not finished. Two-phase locking requires that T1 hold all its locks until it completes the commit. Thus T2 cannot acquire either lock, and thus can't read either value.

12. [7 points]: Suppose T2 has just executed its first line, but has not tried to commit. If the transaction system uses optimistic concurrency control (e.g. FaRM), can the first line of T2 see x=10 and y=2? How, or why not?

Answer: Yes. OCC allows T2 to read whatever values x and y have, even if T1 is executing or committing and hasn't finished. T1 might be in the middle of committing, and have updated x but not y; this would cause T2 to see x=10 and y=2. However, T2 would not be allowed to commit.

VI FaRM

Ben Bitdiddle reads the FaRM paper ("No compromises: distributed transactions with consistency, availability, and performance"). Ben thinks the Figure 4 commit protocol looks too much like two phase commit with pessimistic locking. He decides to get rid of the locks, and use only version numbers. His new commit protocol has no LOCK phase, but instead uses VALIDATE for written objects as well as read objects. Note that FaRM causes a transaction to first read every object that it writes (in order to fetch the version number).

Here is Ben's modified commit protocol:

Phase 1, VALIDATE: The coordinator uses one-sided reads to re-fetch the version numbers of all objects used in the transaction, and aborts if any version has changed since the object was first read by the transaction. (The same as FaRM's VALIDATE in Figure 4 / Section 4, but used for written objects too.)

Phase 2, COMMIT BACKUP: The same as FaRM.

Phase 3, COMMIT PRIMARY: The coordinator sends the primary the updated object contents in this message. The primary copies the new object content to the object's location in memory and increments its version number.

Phase 4, TRUNCATE: The same as FaRM, except that the backups don't unlock.

13. [8 points]: Assume there are no computer or network failures. Describe a specific situation in which transactions would yield incorrect results with Ben's modified commit protocol.

Answer: If there are two concurrent transactions that increment the same object, one of the increments may be lost even though both transactions claim to commit successfully. They'll both read the same value (say 10) and version (say 100), then VALIDATE at the same time and see the version is still 100, then both send COMMIT PRIMARY, both of which cause the primary to set the object's value to value 11. In this situation it's the real FaRM protocol's exlusive lock that forces one of the transactions to abort.

VII 6.824

14. [1 points]: Lab 2 is challenging. If you ran into significant difficulties, what were they? What can we do to avoid these difficulties for future generations of 6.824 students?

Answer: Debugging: 28x General (subtle bugs); 3x Logging; 2x Locking; 13x Test cases/testing; 13x Tips on optimization/performance; 11x Not following details in fig2; 10x GO tips; 9x More time for labs; 7x Make Jon's blog required reading; 5x Synchronization/concurrency; 5x Overall design; 5x More office hours; 4x Have Go guest lecture earlier; 3x Documentation; 3x Edge cases; 3x Dependencies between labs; 3x Debugging tips; 2x Publish raft solution after due date; 2x More skeleton code; 1x Make lab 1 harder to prepare for lab 2; 1x Add implementation assignments for papers; 1x Need detailed instructions; 1x Suggested read: Colin Scott's thesis on testing

15. [1 points]: Which papers should we definitely keep for future years?

Answer: 94x Raft; 51x FaRM; 46x ZooKeeper; 38x MapReduce; 25x GFS; 16x VM-FT; 12x Spinnaker; 11x R*
Add: 2x Paxos; 1x On optimistic methods for concurrency control; 1x Ivy; 1x TxCache; 1x Algorand; 1x MaaT

16. [1 points]: Is there any paper we should delete?

Answer: 33x R*; 25x Spinnaker; 21x VM-FT; 10x FaRM; 7x MapReduce; 6x GFS; 6x ZooKeeper

End of Exam I