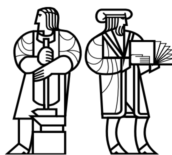


Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.824 Distributed System Engineering: Spring 2016

Exam II

Write your name on this cover sheet. If you tear out any sheets, please write your name on them. You have 120 minutes to complete this quiz.

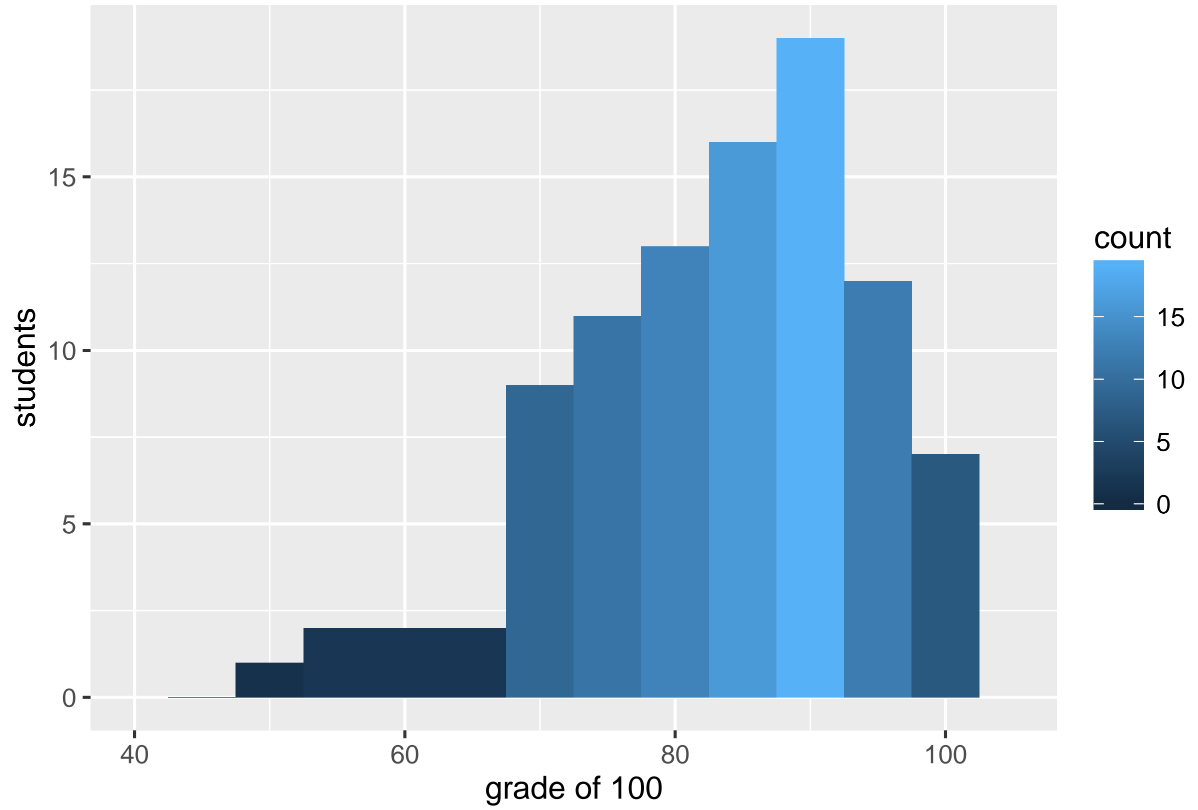
Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, write down any assumptions you make. Write neatly. In order to receive full credit you must answer each question as precisely 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 (18)	II (6)	III (24)	IV (6)	V (12)	VI (12)	VII (6)	VIII (13)	IX (3)	Total (100)

Name:

Grade histogram for Exam 2



$$\begin{aligned} \text{max} &= 100 \\ \text{median} &= 84 \\ \mu &= 82.9 \\ \sigma &= 11.2 \end{aligned}$$

I Bayou

Alice and Bob are siblings. They have trouble getting dressed in the morning because they often choose the same color clothes, which makes them look silly. They decide to solve this problem using an application written with Bayou. Its database has two entries, one for Alice and one for Bob, each entry containing the color assigned to each by the system. On each of their PDAs, the application displays both database entries (if they have values), and allows each of them to specify two colors: a first choice and a second choice. The idea is to assign each their first choice as long as the other hasn't chosen the same color; and otherwise to assign the second choice. Each of them can only specify color choices once.

Their initial prototype works for just a single day. It generates a Bayou write operation that assigns the first choice if the other person's database entry doesn't already contain that color, and otherwise assigns the second choice. Here's what the write operation looks like (modeled on the paper's Figure 3):

```
// who is Alice or Bob, first and second are red or blue.
Choose(who, first, second):
  if who == "Alice":
    other = "Bob"
  else:
    other = "Alice"
  update = { database[who] = first }
  dependency_check = { database[other] != first }
  merge = { if database[other] == first:
              database[who] = second
            else
              database[who] = first
          }
```

> **1. [6 points]:** Is it possible for the application ever to show Bob on his PDA that he's been assigned red and also show Alice on her PDA that she's been assigned red? Briefly explain your answer.

Answer: Yes, if they both chose red as the first choice but their PDAs haven't performed anti-entropy.

2. [6 points]: What stable outcomes are possible if, at the same time, they both choose first=red and second=blue?

Answer: One gets red, the other gets blue.

3. [6 points]: Suppose Bob chooses first=red and second=blue, and then they synchronize their PDAs so that Alice sees Bob's assigned color, and then Alice also chooses first=red and second=blue. After that, Alice synchronizes with the primary, and then Bob synchronizes with the primary. What stable outcomes are possible, and why?

Answer: Bob gets red, Alice gets blue. Since Alice first synchronized with Bob, her log contains Bob's update, and her entry must have a later timestamp.

II Wormhole

Figure 3 of *Wormhole: reliable pub-sub to support Geo-replicated Internet Services* by Sharma et al. describes Wormhole's plan for multiple-copy reliable delivery (MCRD). In MCRD, Wormhole stores the datamarkers for a stream of updates in ZooKeeper. Ben wonders what would go wrong if the datamarkers weren't stored in Zookeeper. Specifically, he considers the following alternative design for the scenario described in Figure 3 of the paper:

- Publisher P2 stores a copy of Publisher P1's datamarkers.
- Periodically, Publisher P1 sends its datamarkers in an RPC to Publisher P2 so that P2 can update its copies.
- If Publisher P2 doesn't receive any updates from P1, it assumes that P1 has failed and starts publishing.

4. [6 points]: What guarantees made by Wormhole would be violated by Ben's alternative design?

Answer: The in-order guarantee would be violated in the case of split-brain scenario: if there is a network partition both P1 and P2 will start publishing, and an application may receive updates from both P1 and P2, out of order.

Update: The Wormhole paper guarantees in-order delivery of updates. However, after further review of the paper, we found that this is only the case for future updates (i.e., a flow can never skip ahead by delivering a later update before an earlier one). It turns out that the scheme we proposed in the exam can never skip ahead, and thus does not violate Wormhole's definition of an in-order delivery guarantee.

III Existential Consistency and Linearizability

Recall the paper *Existential Consistency: Measuring and Understanding Consistency at Facebook* by Lu *et al.* In Section 2.2, the paper defines a linearizable execution as one for which there exists a total order over all operations in the system; the order must be consistent with the real-time order of operations; and each read of a key must see the value written by the most recent write of the same key in the order.

> **5. [6 points]:** Consider the replicated storage system described in the paper's Section 2.1 and Figure 1. There are no failures. Describe how the system could yield results that are not linearizable.

Answer: Client C1 in Region A reads key x and sees value v1, and x/v1 is cached in Region A. Client C2 in Region B writes key x with value v2, and the write completes. But the invalidates are asynchronous and haven't yet reached the cache in Region A. Client C3 in Region A reads the old x/v1.

Suppose there are two clients (C1 and C2) of a key/value storage system. At about the same time, the two clients do the following:

```
C1:
  put("x", "red")

C2:
  if get("x") == "":
    put("x", "green")
```

There is no other activity in the system. Key "x" has an empty string as its initial value.

> **6. [6 points]:** If the system is linearizable, is it possible for the final value (after both clients have finished) of "x" to be "green"? If yes, write down a total order of operations that proves that there a linearizable execution resulting in "green". If a linearizable execution can't result in "green", explain why not.

Answer: The following sequence is a linearizable execution and results in the outcome: C2 reads "", C1 writes red, C2 writes green.

Consider the Principled Consistency Analysis described in the paper's Section 3. Suppose there are three clients. They do the following:

```
C1:
  put(``x'', true)

C2:
  if get(``x'') == true:
    put(``y'', true)

C3:
  if get(``y'') == true and get(``x'') != true:
    put(``z'', true)
```

C1 executes first; a few seconds after C1 finishes, C2 executes its code; a few seconds after C2 is finished, C3 executes its code. There is no other activity in the system. All the keys start with initial value false. We're not sure if the storage system is linearizable or not.

After execution is finished, "z" has the value true.

Principled Consistency Analysis traces only a small subset of the keys in the system. Suppose it traces just one of "x", "y", or "z", but not the others. For each key, indicate whether Principled Consistency Analysis would flag the execution as *not* linearizable if it traced just that key.

7. [6 points]:

```
Just "x":  Linearizable / Not Linearizable

Just "y":  Linearizable / Not Linearizable

Just "z":  Linearizable / Not Linearizable
```

Answer: If z = true, then the trace for x is WT, RT, RF. For y the trace is WT, RT. For z the trace is Wz. The trace for x indicates a linearizability problem, because a later read returns a different value, even there are no writes between the two reads. But tracing only y or z doesn't provide evidence of a problem.

Recall from the PNUTS paper (*PNUTS: Yahoo!'s Hosted Data Serving Platform* by Cooper et al.) that it provides per-record time-line consistency for its operations. The Existential Consistency paper calls this per-object sequential consistency. For the same scenario as the previous question would the Principled Consistency Analysis flag the execution as *not* providing time-line consistency?

8. [6 points]:

```
Just "x":  Time-line / Not Time-line

Just "y":  Time-line / Not Time-line

Just "z":  Time-line / Not Time-line
```

Answer: Time-line for all three. For the tracing x case, the three clients could be reading from different replicas, and maybe C3's replica hasn't seen the put(x,true) yet.

IV Spark

You want to analyze word use in a huge collection of old 6.824 lecture notes. You're using Spark on a cluster of 100 machines (see *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*, by Zaharia *et al*). The lecture notes are stored in a big text file in the HDFS file system. You run the following Spark program to find out how many times each word is used in the lecture notes, and to print the answer for "consistency":

```
1   lines = spark.textFile(input_file)
2   words = lines.flatMap(txt => txt.split(' '))
3   kvwords = words.map(w => (w, 1))
4   counts = kvwords.reduceByKey((x,y) => x+y)
5   counts.lookup("consistency")
```

Line 1 causes each of the 100 machines to read a part of the input file and split it into lines. Line 2 splits each line into words; its output is an RDD of words. Line 3 produces an RDD with the key/value pair (w,1) for each word w. Line 4 adds up the "1" values for each word, yielding an RDD of key/value pairs, one for each word, where the value is the word's count. Note that there are no calls to `persist()`.

Suppose the Spark job is running, and all 100 machines have just finished their part of the work for line 3. At that point one machine crashes. Spark will have to do some work to recover the part of `kvwords` that was stored in the crashed machine's memory.

Then, just after all machines have finished the work for line 4, the machine holding the word count for "consistency" crashes. Spark will have to do some work to recover the part of `counts` that was stored in the crashed machine's memory.

9. [6 points]: Which crash requires more work (computation and network communication) to recover from? Why?

Answer: Recovering counts is more expensive than recovering kvwords. Each partition of kvwords depends only on computations made on the corresponding partition of the input file, so only the work done by the failed machine has to be repeated. Each partition of counts, in contrast, depends on pieces of kvwords from *every* machine (i.e. the reduceByKey is "wide"). So, in order to recompute the failed machine's part of counts, data must be fetched over the network from every machine.

V Borg

As described in the paper *Large-scale cluster management at Google with Borg* by Verma et al., Borg may preempt tasks (e.g., a Map task) from a job (e.g., a MapReduce application with m Map tasks and r Reduce tasks). Borg puts a pre-empted task back in the pending queue, and re-executes it later.

10. [6 points]: Ben worries that Borg's pre-emption might lead to serious waste of CPU time due to re-execution. He is implementing a MapReduce application, and wonders whether splitting the work into lots of tasks or just a few tasks is better for avoiding this waste. Explain to him what the right choice is.

Answer: Lots of tasks. Then re-executing a task doesn't waste much work, since tasks are short.

11. [6 points]: Why would pre-empting a MapReduce master task be a particularly bad idea? How does Borg avoid doing this?

Answer: Preempting a master task would cause the re-execution of the whole MapReduce, which would be wasteful. By making the master high priority, Borg is less likely to preempt it.

VI Chord

Table 1 in the paper *Chord: a scalable peer-to-peer lookup service for internet applications* by Stoica et al. shows the state that each Chord node maintains. One of the entries is the *predecessor* field. Ben suggest modifying the Chord lookup function in Figure 4 to walk either clockwise or counter-clockwise around the ring. If the key *id* is closer to *n.predecessor* on the ring than *n.successor* and any *n.finger[i]*, then lookup returns *n.predecessor* (going counter-clockwise around the ring). Otherwise, lookup works as before (going clockwise around the ring).

12. [6 points]: Assume a static ring with no nodes joining and leaving, and no failures. Further assume that *n.successor*, *n.predecessor*, and all fingers are set correctly. Does the modified lookup function break Chord's performance guarantees? Briefly explain.

Answer: Yes. If the key is in the quarter of the ring preceding n, then the lookup will walk counter-clockwise around the ring one node at a time. Thus, lookup takes $\mathcal{O}(N)$ hops.

13. [6 points]: Consider a dynamic ring where nodes may join, leave, and fail. Does the modified lookup function break Chord's correctness guarantees? Briefly explain.

Answer: This question is imprecise, but the intended answer is as follows: returning the predecessor runs the risk that the caller will walk off the ring to a node that is in the process of joining. That node may have another node hanging off it that is in the process of joining, and so on. If the key is between two nodes that are joining, lookup may return key "not found", even though the node that stores the key is part of the ring.

VII Bitcoin

Consider the Bitcoin protocol as described in the paper *Bitcoin: A Peer-to-Peer Electronic Cash System*, by Satoshi Nakamoto.

14. [6 points]: Suppose one modified the Bitcoin protocol so that mining only occurs when there are new transactions. If there are no transactions waiting to be placed in a block, then the peers (and miners) do nothing. This change might save some electricity. Describe an attack that this change would make easier than in unmodified Bitcoin.

Answer: An attacker can race ahead with a longer fork while main pool of peers is inactive. This allows double-spending, because the attacker can now put a transaction in the current chain, wait for it to be confirmed, and then announce their (longer) fork. This longer fork will become the favored one, and the previous transaction will be invalidated.

VIII Lab 3

15. [7 points]: Ben Bitdiddle notices that his Get operations are too slow. Given that they don't actually modify state, he figures he'll be safe if leaders service them immediately without putting them through the log. Describe a scenario in which this change causes incorrect behavior.

Answer: It's no problem if the leader never changes. But if server S1 receives a Get, and has just lost the leadership, but doesn't realize it, and the new leader has just served a Put, then S1 may reply with stale data.

16. [6 points]: Ben is having some trouble passing tests and he asks you for help. You notice that, to avoid submitting too many operations to the Raft log, Ben is checking for duplicate requests in his RPC handlers but *not* when applying operations from the apply channel. Could this cause an operation to be committed twice? If not, explain. If so, please give a sequence of events exhibiting this bad behavior.

Answer:
Server 1: Become leader.
Client 1: Send op 1 to server 1.
Server 1: Accept op 1 into log. Propagate to server 2. Crash before committing op 1.
Server 2: Become leader.
Client 1: Send op 1 to server 2.
Server 2: Accept op 1 into log. Commit the op 1 received from server 1. Commit op 1 again.

IX 6.824

17. [1 points]: If you could make one change to 6.824 to improve it (other than firing the lecturers and dropping the quizzes), what would you change?

Answer: No reading questions due at 10pm — maybe late policy (7x). More instructions and guidance on labs, especially lab 2 (7x). More guidance on coding/debugging with Go & locks (6x). Post answers to reading questions after deadline (5x). Give students working lab after deadline (5x, but 2x disagreed). Break up labs into smaller deadlines (4x). Diversity in labs: not just Raft (4x). More labs (3x). Paper discussion groups (2x). Lab testing service (2x). Fewer papers (3x). Add recitations (2x). Make lab tests more incremental, less all-or-nothing (2x). Better mentoring on labs (industry/TA code reviews, checkins) (2x).

18. [1 points]: Which papers should we definitely keep for future years?

Answer: 34x Chord, 31x BitCoin, 27x Spark, 26x Dynamo, 25x Raft, 19x Borg, 18x Wormhole, 14x Bayou, 14x PNUTS, 12x Existential Facebook, 6x AnalogicFS, 6x FaRM, 3x Map/Reduce, 3x Zookeeper, 1x Treadmarks, 1x Tail at Scale.

19. [1 points]: Is there any paper that you think we should delete?

Answer: 15x Existential Facebook, 13x BitCoin, 12x Wormhole, 11x Bayou, 11x Borg, 9x AnalogicFS, 6x PNUTS, 5x Dynamo, 4x Spark, 4x Chord, 3x Map/Reduce, 2x FaRM, 2x Treadmarks, 1x Tail at Scale.

End of Exam II