Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.5840 Distributed System Engineering: Spring 2024

Exam II

Please write your name on the bottom of each page. You have 120 minutes to complete this exam.

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.

The maximum number points available is 74.

Gradescope E-Mail Address:

Name: ____________________

Grade statistics for Exam 2

$$\begin{aligned} \text{max} &= 74 \\ \text{median} &= 60 \\ \mu &= 59.61 \\ \sigma &= 9.13 \end{aligned}$$

Name: ____

I Spanner

The intelligent computer HAL is using Spanner (as described in *Spanner: Google's Globally-Distributed Database* by Corbett *et al.*) to store data. HAL notes that read/write transactions are being slowed down by Spanner's commit-wait mechanism (see Section 4.2.1). HAL disables commit-wait in his Spanner installation; as a result, everything works just as described in the paper except that the coordinator leader does *not* wait until the timestamp s is guaranteed to be in the past.

HAL uses just these three transactions:

```
T1:
  X=1
  Y=1

T2:
  X=22
  Y=22

T3:
  print X, Y
```

Initially, database records X and Y both have value 0. X and Y are in different Spanner shards, managed by different Paxos groups. T1 and T2 are read/write transactions; T3 is a read-only transaction.

HAL starts T1; waits for Spanner to say that T1 has completed; starts T2, waits for Spanner to say that T2 has completed; then starts T3 and observes T3's output.

1. [5 points]: Which outputs from T3 are possible? (For each statement, circle True or False.)

True / False : 22, 22

True / False : 1, 1

True / False : 1, 22

True / False : 0, 0

Answer: 22,22, 1,1, and 0,0 are all possible; 1,22 is not. Omitting commit-wait means that either or both of T1 and T2 might commit with time-stamps later than the time-stamp that T3 chooses, so T3 might see the result of either T1 or T2, or neither. T3 can't see 1,22 because both T1 and T2 do both their writes at the same timestamp, so T3 will either see both writes of one of the transactions, or neither.

Name: ______________________________

II Chardonnay

Consider the paper *Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases*, by Eldeeb *et al*.

A read/write Chardonnay transaction reads database record A, then reads B, and then writes C. The system is busy with other read/write transactions at the same time, some of which might also use A, B, and/or C.

2. [4 points]: In which situation will Chardonnay's "dry run" mechanism yield the most benefit? (Circle the single best answer.)

* A is hot, B is cold.

* A is cold, B is hot.

* A is cold, B is cold.

* A is hot, B is hot.

"Cold" means used rarely. "Hot" means used by many transactions.

Answer: Only the first answer (hot, cold) is correct. Chardonnay's dry run mechanism helps avoid situations in which a transaction holds the lock for a record that other transactions need, while waiting to read a record from the disk. This situation arises when a read/write transaction uses a hot record followed by a cold record.

Name: ____________________

A system that uses Chardonnay issues just these three transactions:

```
T1:
  X=1

T2:
  Y=1

T3:
  print X, Y
```

Initially, both database records (X and Y) start out with value 0. X and Y are in different ranges. T1 and T2 are read/write transactions. T3 is a read-only transaction (described in the paper's Section 6). T3 does *not* use the the waiting idea described in the last paragraph of Section 6.2.

One client starts T1. After T1 completes, another client starts T2. After T2 completes, a third client runs T3.

This version of Chardonnay has a bug somewhere in its code, causing T3 to print the incorrect output 0,1.

3. [4 points]: Which of the following bugs is the most plausible explanation for T3 printing 0,1? Circle the single most correct answer.

* The epoch server is stuck: it always returns the same epoch number, and never increases it.

* The epoch server is incrementing too quickly: more than once per 10 milliseconds.

* The epoch server is working correctly *except* it gave T2 an epoch that was too small.

* The epoch server is working correctly *except* it gave T2 an epoch that was too large.

Answer: The third answer is correct. 0,1 is not a correct output because serializability requires that if T3 observes the results of T2, and T1 finished before T2 started, then T3 is required to also see the results of T1. If the epoch server gives T2 an epoch that's less than T1's epoch, and T3 and T1 run in the same epoch, then T3 will see T2's Y=1 but not T1's X=1.

Name: ______________________________

III FaRM

Consider the following statements about FaRM as described in *No compromises: distributed transactions with consistency, availability, and performance*. For each statement, circle True or False.

4. [8 points]:

True / False : Because FaRM uses primary-backup replication for a region (instead of Paxos), FaRM must reconfigure to remove a failed replica before FaRM can continue to use the region.

True / False : FaRM can use short leases (10ms by default) because it has communication and scheduling optimizations to renew leases quickly.

True / False : A transaction that modifies only one object will never abort.

True / False : Read-only transactions require only the validate step of the Commit phase in Figure 4.

Answer: True, True, False, True. The first statement is true because FaRM requires a response from all replicas, thus it must reconfigure to remove the failed replica before it can continue with the affected shard. The third statement is false because another transaction may modify the one object causing this transaction's validation phase to fail (because the other transaction will have incremented the object's version number).

Name: ____________________

IV Ray

Consider the following Ray program, which creates a `sqrt_task` task for each number in the list `mylist`. The creation yields a DFut and the caller waits for the tasks to complete by calling `get` on each future. The code is as follows:

```
# A call to sqrt_task yields a DFut
@ray.remote
def sqrt_task(n):
  # sqrt is a python function, which returns the square root of its argument
  return sqrt(n)

def sqrts0(n_list):
  # start tasks and collect futures
  l = [ ]              # list holding DFuts
  for i in n_list:     # iterate over list of numbers
    l.append(sqrt_task(i))

  r = [ ]
  for f in l:
    r.append(get(f))   # collect the result

  return r

print(sqrts0(mylist))  # invoke sqrts0 with a list of numbers and print result
```

Assume Ray behaves in the way described in *Ownership: a distributed futures system for fine-grained tasks* by Wang et al., and Ray is running on a cluster of computers.

5. [4 points]:

Will the `sqrt` computations complete in the order that `sqrts0` appends to `r`? (Briefly explain your answer)

Answer: No. The `sqrt_tasks` run concurrently with each other, and may finish in an arbitrary order. All that is guaranteed is that the task has finished executing (at least once) by the time `get(f)` returns.

Name: ____________________

Alyssa creates a function `sqrts1` whose body is the same as `sqrts0`, but is declared as a remote task. She then modifies the program to invoke many `sqrts1`'s, each with a large distinct, non-overlapping slice of the number list. The code is as follows:

```
@ray.remote
def sqrts1(n_list):
  ...
  # same code as sqrts0
  ...
  return r

f0 = sqrts1(mylist[...])
f1 = sqrts1(mylist[...])
f2 = sqrts1(mylist[...])
...

print(get(f0))
print(get(f1))
...
```

6. [4 points]:

Ben is worried that the above program creates so many `sqrt_tasks` tasks that Ray will be bottle-necked by managing the tasks and the futures they yield. Briefly explain why Ray can manage many tasks in parallel for the above program?

Answer: The worker machine that invokes `sqrts1(...)` is the owner of the metadata for the value returned by each `sqrts1` call. The many workers that execute `sqrts1()` each independently own the metadata for their `sqrt_task`'s, resulting in no one machine being required to manage all the `sqrt_tasks`.

Name: ______________________________

V Memcache at Facebook

Ben Bitdiddle runs a web site. Ben reads the paper *Scaling Memcache at Facebook* by Nishtala et al., and thinks that the design is too complex. So Ben decides to ignore the paper's design: he *doesn't* use leases, mcrouter, pools, etc. Ben uses *only* the mechanisms described below.

Ben has just a single region, with some web servers, some memcache servers, and a single database server. Ben programs each of his web servers to use the following client code to read and write data:

```
read(k):
  if v = memcache_get(k) succeeds
    return v
  else
    return database_get(k)

write(k, v):
  database_put(k, v)
  memcache_put(k, v)
```

Note that `read()` does not insert anything into memcache, and note that `write()` always inserts the new data into memcache, whether it was already cached or not. Ben knows this may be wasteful, since it may cause memcache to cache data that's never read, but he doesn't mind.

7. [5 points]: Sadly, Ben sees that `read()`s sometimes return stale data for a long time after the `write()` of a newer value has succeeded and returned. Explain how this could happen.

Answer: If there are concurrent writes by different clients to the same key, the calls to `database_put()` may execute in a different order that the calls to `memcache_put()`, so that memcache and the database end up with different values. This condition can persistent for a long time: until the next time a client writes the same key.

Name: ____________________

VI Lab 4

Ben implements the RPC handlers and the applier in Lab 4 as follows. The RPC handlers for `Get`, `Put`, and `Append` take the following steps:

- **A.** Submit a command to the Raft library via `Start`. The command includes the client ID, request ID, operation type, and arguments.
- **B.** Loop to wait until the reply for that command to show up in the reply table, which maps from client IDs to the replies of clients' latest requests. Each reply contains the request ID and the result to that request. If Raft's leadership changes during the loop, return `ErrWrongLeader`.
- **C.** Return the result stored in the reply table.

The applier detail is irrelevant to this question and is shown on the next page.

8. [4 points]:

Ben observes that `Get` does not modify the application state. He changes `Get`'s RPC handler to read the key-value table and return immediately to the client the result. Does this implementation preserve linearizability? (Briefly explain your answer.)

Answer: No. `Get` could return a stale result if Raft the leadership changes. For instance, if a client submits an `Append` to the old leader and succeeds, and then submits a `Get` to the new leader, the `Get` result could miss the appended value if the new leader handles the `Get` before applying the `Append`.

Name:

The applier takes the following steps:

D. Read a command from the apply channel.

E. De-duplicate the command with the reply table: if the request ID in the reply table for the client is greater than or equal to that in the command, then skip the command.

F. Apply the command and insert the result to the reply table.

9. [4 points]:

Separately from the previous change, Ben modifies his implementation to perform de-duplication early in the RPC handlers. Concretely, he removes **step E** in the applier, and adds an additional step at the start of the RPC handlers (i.e., before **step A**) as follows:

If the request ID in the reply table for the client is greater than or equal to that in the RPC arguments, return the result stored in the reply table.

Does this implementation preserve linearizability? (Briefly explain your answer.)

Answer: No. An operation could be applied twice if the client re-sends it before the first RPC is applied.

Name: ____________________

VII AWS Lambda

Consider the guest lecture about the paper *On-demand container loading in AWS Lambda* by Brooker et al. For each of the following statements, indicate whether it is true or false.

10. [8 points]:

True / False : AWS Lambda is attractive to customers because it allows them to run cloud computations without having to provision a machine.

True / False : Many containers of AWS Lambda customers don't contain unique chunks because customers upload the same container multiple times.

True / False : AWS Lambda may deduplicate popular chunks less than unpopular chunks.

True / False : AWS Lambdas use LRU-K to ensure that if many infrequently-used Lambdas are running at the same time, they don't evict the chunks of frequently-used Lambdas.

Answer: True, True, True, True. The third option is true because AWS does this to reduce the blast radius of popular chunks (see Section 3.3).

Name: ______________________________

VIII Boki

Consider Figure 6(a) in *Boki: Stateful Serverless Computing with Shared Logs* by Jia and Witchel. The left column describes how Boki makes the execution of a workflow of serverless functions with database side-effects exactly-once.

Alyssa notices that if Boki reruns a workflow it will append a record to the workflow's LogBook, even if an append of an earlier failed execution already logged the record. Alyssa proposes to change the pattern of append-read to read-append-read: that is, she modifies Boki to read before an append to see if the append already logged its record; if so, it uses the first value returned by the read and skips the subsequent append and read. (If not, Boki executes as before, doing an append followed by read.)

For example, Alyssa changes `write` as follows:

```
def write(table, key, val):
  tag = hashLogTag([ID, STEP])
  # first read
  rec = logReadNext(tag: tag, minSeqnum: 0)
  # if no record, then append and read again
  if rec == None:
     logAppend([tags: [tag], data: [table, key, val])
     rec = logReadNext(tag: tag, minSeqnum: 0)
  rawDBWRITE(...)   # same call as before
  STEP = STEP + 1
```

11. [5 points]:

Alyssa runs one workflow on her modified Boki. The workflow crashes during its execution and then restarts from the beginning and completes. With Alyssa's modification will `write` preserve exactly-once semantics? (Briefly explain your answer.)

Answer: It will preserve exactly-once semantics. In the case that `logReadNext()` returns something non-None initially, it will *always* return that same log record. So even if `write()` did a `logAppend()`, the final `logReadNext()` would have the same value as the `logReadNext()` that is executed before `logAppend()`.

Name: ____________________

IX SUNDR

Consider the straw-man design in the paper *Secure Untrusted Data Repository (SUNDR)* by Li *et al*.

Users A, B, and C share a SUNDR server. The server may be malicious, though the server does not know any of the private keys. User A creates a new file `aaa` in the SUNDR file system. After that, user B looks for file `aaa`, but does *not* see the file. After that, user C creates a new empty file `ccc`.

There is no client activity other than what is described here. None of the stronger consistency ideas from the paper's Section 3.2 are in use. All three users are honest and run correct SUNDR client software.

All three users now use the `ls` command to check whether they can see file `ccc`. All three users' client SUNDR implementations report that the data they receive from SUNDR passes all validity checks. Nevertheless, a malicious SUNDR server can cause a number of different outcomes.

12. [6 points]: What combinations are possible for which users can see `ccc`? For each statement, circle True if the SUNDR server could cause the indicated results, and False if not.

True / False : All three users can see `ccc`.

True / False : Only A and B can see `ccc`, but not C.

True / False : Only A and C can see `ccc`, but not B.

True / False : Only B and C can see `ccc`, but not A.

True / False : Only C can see `ccc`, but not A or B.

True / False : None of the users can see `ccc`.

Answer: The correct answers are A and C but not B, B and C but not A, and only C. We know that the server has forked A and B from the fact that B cannot see aaa. So A and B have seen different operation histories, and each has appended an operation to the history it saw, and remembered that operation. Thus, when C asks the server for the current history (before C creates ccc), the SUNDR server can show C A's fork of the history, B's fork, or perhaps the history as of before A's creation of aaa. As a result, after C creates ccc, ccc will be visible to A (but not B), to B (but not A), and to C alone, respectively.

Name: ____

X PBFT

Consider the PBFT protocol as described in the paper *Practical Byzantine Fault Tolerance* by Castro and Liskov.

13. [5 points]:

PBFT chooses the primary for a view deterministically based on the view number. What could go wrong if PBFT were to use Raft's voting algorithm to select a primary for a view? (Briefly explain your answer.)

Answer: Raft's voting algorithm does not result in a single leader-per-term under byzantine faults. Consider a 7 node system with 2 Byzantine nodes. The nodes that vote for A for term T are A, B, C, D, and allow A to conclude it is leader. The nodes that vote for D for term T are E, F, G, D, and allow D to conclude it is leader. Of these, only D is Byzantine and has equivocated by voting for both A and D. All the other nodes may vote this way while acting non-byzantine. This results in two primaries for a single term and violates the assumptions that the rest of pbft builds on.

Name:

XI Bitcoin

Section 4 of Nakamoto's Bitcoin paper explains that the difficulty of mining is determined by the number of required leading zeros in the SHA-256 hash of the block. The paper also says that Bitcoin automatically varies the difficulty of mining (the number of required leading zeros) by observing the recent average rate of new block mining, relative to the target block every ten minutes; if blocks have been generated too quickly, the difficulty is increased; if too slowly, decreased. All honest Bitcoin peers use the same algorithm to determine the difficulty.

Ben dreams of being able to buy tickets to the latest Taylor Swift concert. To obtain the money required, Ben has been running the Bitcoin peer software on his laptop, but he hasn't been earning mining rewards very quickly, because his laptop is only the winning miner very infrequently. Hoping to realize his dream faster, Ben modifies his copy of the Bitcoin peer software so that the difficulty determination algorithm always yields a low difficulty, with the result that his peer can mine new blocks very quickly, often before any other Bitcoin miner produces a given new block in the chain.

14. [5 points]: It turns out that Ben won't actually earn any bitcoins with this scheme. Explain why not.

Answer: Bitcoin peers that run correct software will check that any proposed new block has a hash with the expected number of leading zeros. Those peers are running the correct difficulty-determining algorithm, so they will reject Ben's blocks because their hashes have too few leading zeros.

Name: ______________________________

XII 6.5840

15. [1 points]: Which lectures/papers should we **omit** in future years?

- Spanner **(3)**
- Chardonnay **(22)**
- FaRM **(8)**
- DynamoDB **(7)**
- Ray **(9)**
- Memcache at Facebook **(3)**
- AWS Lambda **(20)**
- Boki **(62)**
- SUNDR **(6)**
- PBFT **(7)**
- Bitcoin **(1)**

16. [1 points]: Porcupine, the linearizability checker used in the labs, comes with a visualizer that displays in a web browser the entire history of client operations, and highlights non-linearizable errors, if any. Circle the one closest to your experience with the visualizer.

- I don't know its existence. **(46)**
- I've used it, but it isn't particularly helpful to me. **(25)**
- I've used it, and sometimes I understand the result but sometimes don't. **(19)**
- I've used it, and it successfully explains to me why the result is non-linearizable. **(7)**
- I've used it, and it significantly improves my debugging experience. **(2)**
- Other (please briefly describe your experience): **(2)**

17. [1 points]: Do you have any feedback for us about 6.5840?

- (7) less dependence between labs, and want to see/use correct Raft for later labs
- (6) labs about systems/papers other than Raft (e.g. distributed transactions)
- (5) better class notes (e.g. slides)

Name: ______________________________

- (4) better debugging tools (e.g. logging, better visualization)
- (4) TA recitation/office hours for papers specifically + exam review
- (4) guidance on how to read papers/which parts of papers
- (4) stronger tests for earlier labs
- (4) clearer answers to lecture questions
- (2) more office hours
- (2) study fewer papers more deeply
- (3) examples or demos of real world systems
- (2) more lab instructions
- (2) liked emails answering pre-lecture questions
- extra lecture for lab 5
- want more fundamental lecture on formal verification
- lecture on CRDTs, seems like a big topic in distributed systems
- update sundr to a newer system
- ethereum paper get higher priority
- bring back ethereum
- want more time on BFT
- extra credit for lab challenge exercises
- skip lab 2
- want all labs due at end of semester, no late penalty
- want questions due night before instead of the minute before lecture
- survey students to determine how hard labs are
- find a better lecture question for bitcoin
- more grading weight on labs
- the amazon papers were a bit vague in describing precise system/algorithm

Name: ____

- really liked guest AWS lectures
- more conceptual checks on new papers
- want more practice Qs for new paper

End of Exam II

Name: ____