Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.824 Distributed System Engineering: Spring 2022

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 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 or communicate with anyone.

I (7)	**II (14)**	**III (14)**	**IV (14)**	**V (14)**	**VI (7)**	**VII (7)**	**VIII (14)**	**IX (7)**	**X (2)**	**Total (100)**

Name:

Submission Website E-Mail Address:

Grade histogram for Exam 2

q2-hist.pdf

max	=	100
median	=	85
μ	=	84.0
σ	=	10.5

I Frangipani

Consider the paper *Frangipani: A Scalable Distributed File System*, by Thekkath *et al*. Four users, α, β, γ, and δ, each with their own workstation, share a Frangipani file system. The file system appears under the pathname `/f`. The file system starts out containing no files and just the empty directory `/f`.

At about the same time, α runs

```
cp /dev/null /f/a
```

and β runs

```
cp /f/a /f/b
```

and γ runs

```
cp /f/b /f/g
```

`cp` is a command that copies one file to another. If the source file doesn't exist, `cp` prints an error message and does *not* create the destination file. `/dev/null` is an empty file, so the first command is a way to create a new file (`/f/a`) containing nothing.

All the above commands complete and indicate that they were successful. There is no other activity, and no failures up to this point.

Now the workstations of α, β, and γ crash!

1. [7 points]: After the crashes, user δ looks at the Frangipani file system, `/f`, from her workstation. What might δ see as the *complete* output of a directory listing of /f? Circle all correct answers.

A. No files.
B. /f/a
C. /f/b
D. /f/g
E. /f/a /f/b

Answer: Only E. δ will definitely see /f/a and /f/b, and might (or might not) see /f/g. The fact that all the `cp` commands completed successfully means that they ran one at a time in the order shown above; each of them was able to read its input and created its output file. Further, the workstations of α and β must have given up their locks, since the subsequent commands needed those locks; this means that the information about the creation of /f/a and /f/b must have been written to Petal, and thus must be visible to δ (perhaps after δ runs recovery for the crashed workstations). On the other hand, γ may have crashed while holding locks, and may never have written any information about /f/g to Petal (not even its log), so δ might not see /f/g.

II Spanner

2. [7 points]: Sections 4.1.4 and 4.2.2 of *Spanner: Google's Globally-Distributed Database* say that in many cases Spanner assigns a read-only transaction a timestamp of TT.now().latest. What useful property does this choice of timestamp have? Circle the single best answer.

A. This timestamp is guaranteed to be less than t_{safe} at every replica.

B. This timestamp is guaranteed to be less than the timestamp of any concurrent read-write transaction.

C. This timestamp is guaranteed to be greater than the timestamp of any completed read-/write transaction.

D. None of the above.

Answer: Only **C** is correct. **A** is wrong because t_{safe} might lag arbitrarily on a Paxos replica that's not part of a majority. **B** is wrong because concurrent transactions might have timestamps in either order. **C** is correct because a read-write transaction doesn't complete until the time of its timestamp is guaranteed to be in the past (this is the commit wait).

3. [7 points]: For a read-only transaction that reads multiple keys stored on different Paxos groups, why wouldn't it be correct to read the most up-to-date version of each of the keys? Please briefly describe an execution in which the results would be incorrect.

Answer: The result might not be serializable if a read-write transaction executes in the middle of the read-only transaction. That is, the read-only transaction might see one value from before the read-write transaction, and the other value as of after the read-write transaction. Then the result may not be equivalent to either one-at-a-time ordering of the transactions.

III FaRM

Consider the FaRM execute/commit protocol in Figure 4 of *No compromises: distributed transactions with consistency, availability, and performance*.

The following two transactions run on different computers that share a FaRM storage system:

```
T1:
  txCreate()              // start the transaction
  ox = txRead(x_oid)      // read object x (an integer)
  txWrite(x_oid, ox + 1) // write object x
  oy = txRead(y_oid)      // read object y (an integer)
  oy = compute using ox and oy for a while
  txWrite(y_oid, oy)      // write object y
  txCommit()              // start Figure 4's commit phase

T2:
  txCreate()
  ox = txRead(x_oid)
  oy = txRead(y_oid)
  ok = txCommit()
  if ok:
    print ox, oy
```

`txRead()` uses RDMA to read an object with a given object identifier into local memory. `ox` and `oy` are local variables. `txWrite()` buffers the written data, which is written to FaRM during the commit phase if the transaction commits. `txCommit()` returns true if the transaction committed successfully, and false if it aborted.

Suppose T1 is in its Execute phase, and has just read object x. At this point, T2 starts executing, on a different computer. Nothing else is happening. There are no failures.

4. [7 points]: Could T1 cause T2 to abort? Explain briefly how this could happen, or why it can't happen.

Answer: Yes, if T1 commits after T2 reads y but before T2 commits, or if T1's lock on y is set when T2 reads y. Then T2's VALIDATE message for y will see an increased version number.

5. [7 points]: Is it possible for T2 to successfully commit? Explain briefly how this could happen, or why it can't happen.

Answer: Yes, if T2 entirely completes before T1 starts to commit.

IV Spark

The paper *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing* by Zaharia *et al.* mentions a distinction between wide and narrow dependencies.

6. [7 points]: Some kinds of dependencies, such as join, are sometimes wide and sometimes narrow. Join is a transformation that brings together records from its two input RDDs that share the same key (so a join's inputs must be RDDs of key/value pairs). Circle the single answer that best explains when join is wide versus narrow.

A. Join is narrow when executed lazily, and wide when forced to materialize by an action such as collect().

B. Join is narrow when its two input RDDs resulted from narrow dependencies, wide otherwise.

C. Join is narrow when both preceding RDDs are partitioned by key in the same way and the join is joining on that key, wide otherwise.

D. Join is narrow when the preceding RDD is persistent, wide otherwise.

E. None of the above.

Answer: C.

7. [7 points]: Circle the true statements (there may be more than one).

A. A wide dependency requires data to be moved across the network, while a narrow dependency does not.

B. A narrow dependency can be executed lazily, while a wide dependency must be executed immediately.

C. A narrow dependency requires input RDD data to be key/value pairs, while a wide dependency's input can be arbitrary records.

D. A narrow dependency is likely to execute more slowly than a wide dependency.

E. If a worker fails, narrow RDD partitions for which it was responsible may have to be re-computed starting with the original inputs.

Answer: A, E

V Memcached at Facebook

Figure 1 of *Scaling Memcache at Facebook* by Nishtala *et al.* illustrates a straightforward way to use memcache as a cache for a database. The left-hand part of Figure 1 shows how an application reads data, and the right-hand part shows how an application writes data. For these questions, assume the application caches database records in memcache, with the same keys that the database uses.

8. [7 points]: Consider the scheme in Figure 1 **without leases** and without any of the paper's other improvements (without McSqueal, without McRouter). Why does Figure 1's write path do the memcache delete *after* the database UPDATE, rather than before? Circle the single best answer.

A. This order guarantees that a read concurrent with a write of the same key won't cause stale data to be cached in memcache.

B. This order prevents a read from seeing the old value after the UPDATE completes.

C. Neither of the above.

Answer: **C** (neither of the above). **A** is incorrect because a read might miss and fetch data from the database, and only "put" the data into memcached after the write is entirely complete. **B** is incorrect because a read might see stale data in memcache between the UPDATE and the delete.

9. [7 points]: Now consider a design consisting of Figure 1 **with leases** (but without any of the paper's other mechanisms). In which situation(s) does the lease mechanism decrease the likelihood of stale data being cached in memcached indefinitely, compared to Figure 1 alone? In all of these scenarios, the read's get misses in memcache. Circle all of the correct answers.

A. When there are two concurrent reads of the same record (but no writes).

B. When the two steps of a write occur between steps 1 and 2 of a read of the same record.

C. When the two steps of a write occur between steps 2 and 3 of a read of the same record.

D. When the three steps of a read occur between steps 1 and 2 of a write to the same record.

E. None of the above.

Answer: C.

VI COPS

Alyssa stores data in COPS (*Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS* by Lloyd *et al.*). The COPS deployment has three replicas, in three data-centers. Alyssa runs code on three clients (C1, C2, and C3), each in a different data-center along with one of the COPS replicas. Alyssa is using COPS (not COPS-GT).

Alyssa's data consists of two key/value pairs: I1, whose value is a string, and I2, whose value is an integer. I1's value starts out as the empty string, and I2's value starts out as zero. Both key/value pairs start out identical at the three COPS replicas.

At about the same time, each of Alyssa's three clients runs some code that calls the COPS client library put and get functions. There's no other activity, and no failures.

```
C1 runs this:
  put("I1", get("I1") + "X")   // append X to I1
  put("I2", get("I2") + 1)     // increment I2

C2 runs this:
  put("I1", get("I1") + "X")   // append X to I1
  put("I2", get("I2") + 1)     // increment I2

C3 runs this:
  print "I2=", get("I2")  // read I2 first
  print "I1=", get("I1")  // then I1
```

10. [7 points]: What output could C3 print? Circle all of the correct answers.

A. `I2=2 I1="XX"`
B. `I2=2 I1="X"`
C. `I2=2 I1=""`
D. `I2=1 I1="XX"`
E. `I2=1 I1=""`
F. `I2=0 I1="XX"`

Answer: A, B, D, F (but not C or E). **A** results from a straightforward one-at-a-time execution. **B** can result if C1 and C2 both read the initial empty string for I1, but one sees the other's incremented I2. **C** cannot occur because COPS's causal machinery ensures that if C3 sees either C1 or C2's write to I2, it must also see that client's previous write to I1. **D** can occur if C3 reads I2 after just C1 executes, but C3 sees I1 after C2's writes have appeared as well. **E** cannot arise for the same reason as **C**. **F** can occur if C3 reads I2 before either C1's or C2's updates appear at C3's replica, and reads I1 after both have.

VII SUNDR

Three client computers (C0, C1, and C2) share files on a SUNDR server. (*Secure Untrusted Data Repository (SUNDR)*, by Li *et al.*). The SUNDR server starts out containing three files, x, y, and z, and their contents all start out as 0.

Client C0 asks the SUNDR server to perform the following three writes, one at a time, waiting for each to complete before proceeding to the next:

```
C0: write "1" to file x
C0: write "2" to file y
C0: write "3" to file z
```

The SUNDR server returns a success indication for each of these writes, and C0 does not detect any problem. After C0 has completed its writes, C1 asks the SUNDR server to perform the following three writes, one at a time, waiting for each to complete before proceeding to the next:

```
C1: write "1" to file x
C1: write "2" to file y
C1: write "3" to file z
```

Again, the SUNDR server returns a success indication for each C1's writes, and C1 does not detect any problem. After C1 has completed its writes, C2 asks the SUNDR server to perform the following three reads, one at a time, waiting for each to complete before proceeding to the next:

```
C2: read file x
C2: read file y
C2: read file z
```

The SUNDR server returns a success indication for each of these reads, and C2 does not detect any problem.

The three clients only communicate with the SUNDR server; they don't talk to each other. You can answer this question with reference to the paper's straw-man protocol (Section 3.1), though that should not affect the answers.

11. [7 points]: Which of the following could C2 see from its three reads?

A. x=1 y=2 z=3

B. x=0 y=2 z=3

C. x=0 y=0 z=0

D. x=1 y=0 z=0

E. x=1 y=0 z=3

F. x=1 y=1 z=1

Answer: **A**, **C**, and **D**. **B** and **E** cannot occur because if the SUNDR server hides one of a client's updates, it must hide all of that client's following updates. **F** cannot occur because the SUNDR cannot forge writes that no client issued.

VIII Bitcoin and Ethereum

Ben runs an online tulip shop where he accepts Bitcoin as payment, 1 BTC for 1 tulip (these are premium tulips). Upon seeing a payment transaction appear in the blockchain, Ben waits for 6 additional blocks before treating the payment as valid and shipping tulips to the buyer by overnight Fedex.

12. [7 points]: Suppose an attacker owns 1 BTC, and the attacker gains control of 80% of the Bitcoin network hash power for a period of 2 hours. Describe how this attacker could cheat Ben's store, such that on the next day, the attacker still has their 1 BTC and also has a tulip.

Answer: The attacker should submit a transaction paying Ben the 1 BTC. In secret, the attacker should mine a fork of blocks from the block *before* the block containing that transaction. After Ben has shipped (i.e. after an hour), reveal the fork. It will be longer than the real fork due to the 80%, so everyone will switch to it. But it doesn't contain the payment to Ben, so the attacker still owns the 1 BTC. To prevent Ben from re-broadcasting the transaction from the shorter fork on what is now the main chain, the attacker should include a transaction in their fork to move their Bitcoin to a different address.

Alyssa runs a virtual tulip shop implemented as an Ethereum smart contract. Her shop sells sells tulip "tokens" for 1 ETH (ETH is the Ethereum currency). A tulip token is a purely virtual item, whose ownership is represented in the state of Alyssa's smart contract; no physical goods are involved. Here's what Alyssa's Ethereum tulip smart contract state and external method look like:

```
// track how many tulip tokens each ETH address holds
private mapping (address => uint256) tulips;

function buy() external payable {
    require(msg.value >= 1 eth); // sender has to transfer >= 1 ETH
    tulips[msg.sender]++; // sender gains a tulip token
}
```

13. [7 points]: Suppose an attacker owns 1 ETH, and the attacker gains control of 80% of the Ethereum network hash power for a period of 2 hours. Could the attacker cheat Alyssa's store, such that on the next day, the attacker still has their 1 ETH and also has a tulip token? Why or why not?

Answer: The attacker cannot cheat. The attacker could produce a longer fork (as for the above Bitcoin question). But if the longer fork omits the ETH transfer, it must also omit the change to the smart contract state that increments the attacker's tulip token count.

IX Lab 3

Alyssa is working on reducing the read latency of her Lab 3 by allowing either a leader or a follower to respond to a Get() RPC, and to do so without putting the operation in the Raft log. Her plan is as follows:

A. Modify the Put() and PutAppend() RPC responses to return the Raft log index where the operation committed.

B. Modify the client Clerk module to track the latest log index received in a Put() or PutAppend() RPC response, and pass the index as an additional lastPutIndex argument to the Get() RPC request.

C. Modify the server code to execute Get() requests with a given lastPutIndex as follows, on either the leader or a follower:

(a) If the local replica has finished processing at least up through index lastPutIndex, then read the data from the local replica's state and return immediately.

(b) Otherwise, wait until the command at lastPutIndex has been processed, and then read the data from the local replica's state.

14. [7 points]: Alyssa claims that her design will order her Get() requests after previous Put()/PutAppend() requests, and this will prevent stale reads. Is Alyssa's modified design correct — does her design still implement a linearizable key-value store? If yes, explain why it's correct. If not, give a scenario in which Alyssa's design results in an execution that violates linearizability.

Answer: Server S1 might be leader, but then lose leadership to another server, which processes some Put()s from other clients. The client sends a Get() to S1, which responds using its (now out-of-date) state. Or the client might send a Get() RPC to a follower, but that follower doesn't know about the most recent Put()s from other clients because it wasn't in the leader's majority. The follower would respond to the Get() with stale data.

X 6.824

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

- Distributed Transactions
- Frangipani
- Spanner
- FaRM
- Spark
- Memcached at Facebook
- COPS
- SUNDR
- Bitcoin
- Blockstack
- Ethereum

16. [1 points]: Do you have any feedback for us about 6.824?

End of Exam II