James Crowell
CS455-Sp22
4/29/22

- This exam has 3 questions, for a total of 100 points.
- The exam answers should be in a single PDF file named final.pdf and pushed to your indvidual GitHub repo in the `final` subfolder.
- The final is due by midnight on 2nd May, 2022.

Question	Points	Score
1	50	
2	10	
3	40	
Total:	100	

1) **Amazon Dynamo Paper Concepts**

Communication

Dynamo uses a “gossip-based protocol”, which is a model apparently based on how epidemics spread. It’s a peer-to-peer system wherein Dynamo nodes start, “choose” a set of tokens which are themselves virtual nodes within a hash space that is consistent between/within nodes, and then map those tokens to the node. Each node that is involved with a read or write to a key in the data store will replicate (“forward”) those operations to other nodes that are in its map set. Mentioned in “Failure Detection”, nodes also appear to be arranged in a Ring formation, and do make use of a Coordinator (as well as “Java NIO channels”, which is another thing I’ll be looking into during the summer; it’s a term I’ve never heard before).

Consistency

Accomplished by way of “object versioning”, and maintained between different replicas by a “quorum-like technique”, culminating in an eventually-consistent storage system, according to the paper. Being less stringent on consistency is explicitly mentioned to increase resource availability (which makes sense, considering requiring a higher degree of consistency is going to slow systems down, either as they authenticate replicated actions and presumably repeat them if they fail, as well as a requirement that to ensure correctness of an answer, the data for that answer must be made available until the certainty of said answer is absolutely correct; achieving both these ends simultaneously is declared infeasible). Dynamo appears to have settled on allowing for all data store updates to reach all data replicas *eventually*.

Fault Tolerance

Strict quorum is explicitly mentioned as being too unforgiving, and given to leading to full system collapse during server/net partition failure. Instead, nodes are programmed to, if they require a consensus of value N, to simply take the first N nodes they encounter that are “healthy”, as enumerated in its preference list. This data is later on shunted back to the originally-intended nodes once they are healthy again, because “crutch nodes” (my term, not the paper’s) store the data they accept in a downed node’s place in a different repository, and said repository is regularly checked for the presence of “crutch data”.

Replication

Data is stored on Dynamo nodes both locally and at successor nodes in the Ring. Apparently the number of hosts to replicate to can be configured per-instance, “N”, leading to data instances existing at the node it is passed to, as well as N-1 other nodes storing copies. The Coordinator controls key replication, and does so in a clockwise direction. Therefore, if a key K is given to node B with a copy parameter of 3, that key value/change is stored on nodes B, C, and D, as a for-instance. This system means each node watches out for an entire sector of the Ring, and it rather reminded me of a RAID array.

Synchronization
Updates are run asynchronously, which is an element of the eventual consistency noted above. The "hinted handoff" system noted in "Fault Tolerance" also facilitates eventual consistency in that crutch nodes can hold onto replications meant for currently-downed nodes until such time as they can be restored and properly written to. The other half of this is Dynamo's anti-entropy protocol, exemplified (I think) by the Ring structure and Merkle trees, which allow nodes to keep track of sets of each other (see the "Communication" section on node hashing sets of nodes to themselves to track) independently, and said sets can be queried internally to keep track of status without requiring participation of the system as a whole. Quote, "Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during anti-entropy process."

2) Top Three Technical Takeaways

- Multicasting: Easily the thing I'm most excited about from this class, and that's saying quite a bit. I'm *moderately* frustrated by the fact that the day after P3 was due, I woke up with a fully-formed multicasting scheme to facilitate cluster communication, including both election protocol and heartbeat sending/detection, which I am very sure will work, but I recognize that the only reason I was able to construct it subconsciously was because of 2.5 straight weeks of working on the project. Anyway, I find multicasting both fascinating and a godsend because direct connection for wide-scale communication is horrendously inconvenient, and therefore slow, and therefore expensive, whereas with MC, we can spit one packet out and have it be "cloned" to every other machine on the network. Granted, no guaranteed delivery, but the answer to that is obviously *SPAM*. I'm very excited to work more with this in the future

- SSL Encryption: We didn't actually do a whole lot with this (or perhaps rather, maybe it's just even simpler than I expected it to be because Java's native SSLSocketFactory library just *does* it), but I've always heard about end-to-end encryption in network security, but never really had to work with it in a project. That felt particularly strange, making systems that paid no heed to security whatsoever; I always assumed that it was because the difficulty of integrating such features would be beyond the scope of the classes we were enrolled in. It seems not so much the case, thanks again to Java's endemic security methods/libraries. I would like to dissect these a little further and see what encryption methods they use to achieve this goal

- Docker: Oh, man, the DevOps unit? Not to cast aspersions, but this class very much did it right, whereas 471, not so much. Being able to create my own little mini-OS's that are tailored to whatever specific needs I have (to the point that making a new container doesn't even natively contain MAKE, for God's sake; I don't know how you *manage* that. I can run APT-INSTALL in these containers, and many of those install processes need MAKE to resolve. Or I thought they did, anyway. I don't know how you'd do them without it), as well as use them to simulate multiple different machines on the same network to mock up distributed systems was just *so* cool. It is a rare thing that I encounter doing schoolwork that I decide to put on my primary leisure rig at home, and Docker was definitely one of them; I can think of all kinds of projects to work on over the summer, assuming I have free time (I think a 9-5 internship will lead to more free time than 16 credits of school enrollment) again.

More than anything, this is perhaps the only non-foreign-language class I've taken where I've been able to say "I was never bored, and there was never a single thing we talked about that I wasn't interested in". Easily my hardest class for this degree. Easily my favorite, too.

Gonna try to start P4 no later than Monday (only a couple tests mean I should be able to just hammer on it for three straight days), but I don't know if I'll pull it off. I'm genuinely shocked at how relieved and exhausted I felt after turning in P3. I might not recharge fast enough, but I'm gonna try!

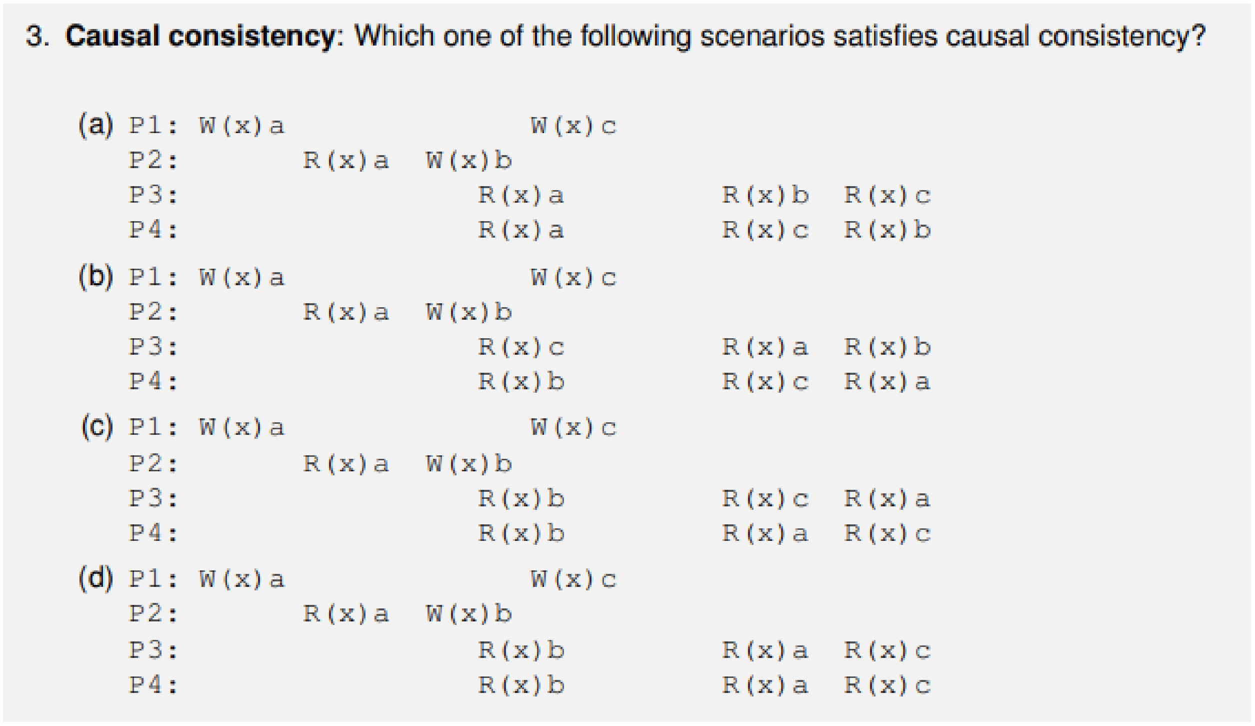
3) Quiz 4

1. Which of the following types of logical clocks is appropriate for implementing totally-ordered multicasting within a distributed system?

(a) Lamport timestamps <-
(b) Both Lamport timestamps and Vector Clocks
(c) Vector Clocks
(d) None of the choices

2. Which of the following statements correctly describes causal consistency?

(a) A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.
(b) None of these.
(c) All writes must be seen by all processes in the same order. Write by the same process are also seen in the order they are performed by a process.
(d) Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes by different processes may be seen in a different order on different machines. Writes by the same process are also considered to be causally related. <-


3. **Causal consistency**: Which one of the following scenarios satisfies causal consistency?

```
(a) P1: W(x)a              W(x)c
    P2:       R(x)a  W(x)b
    P3:                R(x)a          R(x)b  R(x)c
    P4:                R(x)a          R(x)c  R(x)b
(b) P1: W(x)a              W(x)c
    P2:       R(x)a  W(x)b
    P3:                R(x)c          R(x)a  R(x)b
    P4:                R(x)b          R(x)c  R(x)a
(c) P1: W(x)a              W(x)c
    P2:       R(x)a  W(x)b
    P3:                R(x)b          R(x)c  R(x)a
    P4:                R(x)b          R(x)a  R(x)c
(d) P1: W(x)a              W(x)c
    P2:       R(x)a  W(x)b
    P3:                R(x)b          R(x)a  R(x)c
    P4:                R(x)b          R(x)a  R(x)c
```

(This one didn't comfortably C/P into the doc, so I just screencap'd it)
Answer: **A**

4. Which blue pill would you want? Which one of the following statements "ranks" the consistency models from the "weakest" to the "strongest"?
a) Eventual Consistency < Sequential Consistency < Causal consistency
b) Causal Consistency < Eventual Consistency < Sequential Consistency
c) Eventual Consistency < Causal Consistency < Sequential Consistency <-
d) Sequential Consistency < Causal Consistency < Eventual Consistency

5. Consider a quorum-based protocol, where N = 5 and the read quorum is 3 and the write quorum is 3. Which of the following statement is correct about this type of replicated system?
(a) There are no read-write or write-write conflicts. <-
(b) Only read-write conflict is possible.
(c) Both read-write and write-write conflicts are possible.
(d) Only write-write conflict is possible.

6. How many total processes would a system need for Byzantine Agreement such that it can tolerate 3 faulty servers?
(a) 4
(b) 6
(c) 7
(d) 10 <-

7. In the one phase commit algorithm, the coordinator simply tells all participating processes to commit. Which of the following statements about one phase commit algorithm is correct?
(a) There is no problem whether any process crashes or not.
(b) There is no problem as long as a participating process does not crash.
(c) There is no problem as long as the coordinator does not crash.
(d) If a participating process cannot commit, there is no way for it to tell that to the coordinator. <-

8. In the two phase commit algorithm, what action should the participating processes take if the coordinator crashes and all participating processes are in the READY state? (assume that the participating processes have all contacted each other)
(a) Make transition to ABORT state.
(b) I have no clue and I refuse to guess!
(c) Wait for the coordinator to recover. <-
(d) Make transition to COMMIT state.

(Not too sure on this OPC/TPC ones; everything I looked at for #8 here suggested that followers BLOCK until being given a command from the Coord, and that's often marked as a big downside of the algo)

9. Suppose that a given server is available 75% of the time (that is, it is down 25% or 1/4th of the time). How many times do we have to replicate it (not counting the first instance) to increase availability to 99% of the time (that is, it is down 1% or 1/100th of the time)?
(a) 4 <-
(b) 3
(c) 25
(d) 10

(On the one hand, I feel like doubling your server count halving your downtime makes sense. On the other hand, it feels too "easy" of an answer…)

10. The CAP theorem refers to three big concepts that cannot exist simultaneously at a given instant in distributed systems. These three big concepts are:

(a) Communication, Access, Processes

(b) Consistency, Availability, Partition tolerance <-

(c) Countability, Availability, Processes

(d) Communication, Access, Process Replication