CPS 512 midterm exam #2, 11/16/17

Your name please: _____ NetID:_____ Sign for your honor:_____

Part 1. 2P* (50 points). Consider a system for client/server transactions with two-phase locking (2PL) and two-phase commit (2PC), as in the Lab #3 KVClient/KVStore. The following statements apply for any transactions *S* and *T*, any lock *L*, and any participant servers *P* and *Q* in such a system. Are they True or False? Indicate your answer (T or F) in the box at the **left** of each statement. You may comment further in the space on the right if you wish.

T	*T* holds all of its locks at one time.	*A point in time between growing and shrinking phase.*
F	*T* may acquire *L* twice or more as *T* executes.	*No acquire after release.*
T	*T*'s execution might overlap with *S* even if both acquire *L*.	*While at least one does not hold L.*
T	If *S* acquires *L* before *T* acquires *L*, then *S* commits before *T* (in the serial order).	
T	If *S* commits before *T*, and both acquire *L*, then *S* acquires *L* before *T* acquires *L*.	*Required by ACID and serializability.*
T	If *S* and *T* both acquire *L* on *P*, and *S* requests to commit first, then *P* does not vote to commit *T* before *P* receives the outcome of *S*.	*Required by ACID and serializability.*
T	If *P* votes to abort *T* then *T* cannot commit.	
F	If *P* votes to commit *T*, and does not receive the outcome of *T* from the coordinator, then *P* may abort *T* and release its locks.	*Only after it determines the outcome.*
F	If *P* and *Q* are the only participants in *T*, and both vote to commit *T*, then *T* must commit.	
F	If *S* acquires only lock *L*, and *T* acquires multiple locks including *L*, then deadlock is possible between *S* and *T* under 2PL.	

/50
/30
/40
/40
/40
/200

Part 2. More 2P* (30 points). These questions pertain to your implementation for Lab #3.

(a) How does it ensure that a transaction does not read stale data out of its client cache?

This is an exercise in precise explanation, and I was a little harsh here. It is not enough to simply say the transaction holds locks, and/or that it pushes writes to the stores on commit. Topic was covered before exam #1.

(b) How does it detect and resolve deadlocks among transactions?

Abort-on-timeout works, if you take care to release all locks on abort: see (c) below.

(c) When are the locks held by a transaction released?

Don't say the app or client does it when the transaction is done: the app API has no locks, and the client may fail after reporting commit outcome. Stores must release locks upon receiving commit/abort outcome.

Part 3. RSM (40 points). These questions pertain to Replicated State Machine (RSM) primary/backup replication using a Consensus algorithm (e.g., Viewstamped Replication). Failures are fail-stop. All servers are replicas. Feel free to draw.

(a) You are building an RSM system to tolerate up to three simultaneous server failures. How many servers does your system need?

2f+1 → 7

(b) If the network is partitioned AND three servers fail, is it possible for that system to continue to serve requests? Explain or illustrate by describing a scenario.

Three servers fail on the minority side, leaving a non-faulty majority on the other side.

(c) Suppose that at the start of a new view in an RSM system, a single server *S* has a log entry *E* with a higher index (op-number) than any entry on any other server. How could this occur? Illustrate with a scenario.

The partition heals and f servers fail from the majority in the last view, leaving one. This is a variant of "nasty scenario getting nastier".

(d) Could that *E* have committed in some previous view?
This question is asking about *any* scenario with such an E, and not just the scenario you outlined.

Yes. E might or might not have committed: the new view might not know.

(e) Is it safe to preserve that entry *E* in the new view if *E* did not commit in any previous view? Explain or illustrate with a scenario.

*Yes, safe, because whether or not E had committed, **no other conflicting entry could have committed** in that index, or some server in the new view would know of it.*

(f) Suppose a network partition occurs in which a server *S1* has a minority on its side, and a server *S2* has a majority on its side. Can *S1* write new log entries? If so, is it possible that *S1*'s entries have a higher view (term) number than *S2*'s entries? Explain or illustrate with a scenario.

Yes, S1 can write entries (if the old leader survived on the minority side as in "nasty scenario"), but they cannot commit. No, they can't have a higher term number because the minority side cannot advance to a new view, because it cannot muster a majority to elect a new leader. S1 is stuck in the past while the world moves on.

Part 4. Conflicts in RSM and RWN (40 points). Questions (a) and (b) pertain to Consensus RSM systems as in Part 3. The remaining questions pertain to an RWN "sloppy quorum" key-value store in which any of the servers/replicas for a key *K* may coordinate an operation on *K*, following the model of Dynamo.

(a) Suppose that at the start of a new view in an **RSM** system, server *S1* has an entry *E1* in its log, and server *S2* has an entry *E2* that conflicts: *E2* has the same index (op-number) as *E1* but a different operation than *E1*. How could this occur? Illustrate with a scenario. **This question is obviously asking about the sequence of events that produced a pattern of log entries among the voting majority for the leader of the new view.**

As in 3(c), when that partition heals, S1 and S2 may have conflicting entries for some index.

(b) What does the system do with *E1* and *E2* in that case?

Pick the one with the highest view/viewstamp. That will be S2's: the majority side advanced the view to elect a new leader.

(c) Now consider an RWN service with *N*=5, *W*=2, *R*=1. Suppose a partition occurs in which a server *S1* has a minority on its side, and a server *S2* has a majority on its side. Describe a scenario like this in which *S1* can still coordinate and complete a client write request for a key *K*.

Same as 3(c), but now S1 can complete a write if its minority includes at least W=2 members.

(d) Suppose then that *S1* serves write *W1* on *K* and *S2* serves write *W2* on *K*, both during the partition. **After the partition heals**, what value does *S1* or *S2* return for a read on *K*? **This question is obviously asking about the behavior after post-partition repair actions, i.e., after the replicas converge.**

These writes are concurrent and the servers have no basis to choose among them. "Kick it to the app". Dynamo returns both values, unless and until the app writes to the key with a vector timestamp that subsumes the concurrent writes.

(e) Describe a scenario in which a **similar write conflict** might occur in an RWN system with *N*=5, *W*=3, *R*=2. **Here a "similar write conflict" means a scenario with two concurrent writes as in (d).**

A write conflict occurs if there are two concurrent ops on the same key, and at least one is a write. Since N=5 and W=3, a partition alone won't create the scenario. But even without a partition, concurrent writes can occur with no defined order because there is no primary to sequence them. Just suppose that two different servers coordinate writes on K "at the same time", and other servers receive them in different orders. A stale read preceding the second write could also do it.

Part 5. Request Routing (40 points). These questions pertain to the routing of requests to servers. CH is Consistent Hashing (CH) as explained in class. Suppose keys are evenly distributed through the key space and load is (initially) distributed evenly among the keys. The load for a key *K* is given by the number of requests on *K* multiplied by *K*'s mean per-request cost.

(a) CH and Slicer attempt to balance load among the servers. What is your favorite metric to quantify how well the load is balanced? What is the value of your metric when the load is perfectly balanced? **(10 points)**

*Slicer and Dynamo use **max/min** or **max/mean**. Max/min takes into account under-loaded servers and is easier to work with, but max/mean allows you to ignore the substance of the question. Need a **global metric** derived from per-server **load** (work/time), e.g., utilization. If all have the same load then max/min = 1 and max/mean = 1. Response time metrics don't work here.*

(b) Consider CH with 10 buckets (servers) and no virtual nodes (tokens): each server is hashed to a single point on the unit circle (ring). Suppose that the load is perfectly balanced and then we add one server. What is the new expected value of your metric? (Approximate.) **This question is obviously asking for the metric value after rebalancing.**

One of 10 servers splits its load with the 11th, so both see half the load of the others. Max/min = 2.

(c) Consider the same scenario but with two tokens per server. What is the new expected value of your metric after adding an 11th server? Approximate **after rebalance.**

Two of 10 servers split their load with the 11th. It is expected that they each give 25% of their load: half of the load from one of their two tokens. But the new server has 50% of max so max/min = 2.

(d) Consider the same scenario, with 10 servers. Suppose further that there are 1000 keys, and then load for a single key *K* jumps to 100 times the others. What is the new expected value of your metric? **(after rebalancing)** Does it matter how many tokens per bucket?

Does not matter how many tokens. Each server serves expected 100 keys, so the single server with the hot key experiences roughly doubled load: the hot key now has roughly the same load as its other 99. So max/min = 2.

(e) What is the impact of that scenario in (d) on P90, the 90-percentile response time tail? (Approximate.) What does your answer depend on?

Answer depends on #requests R for the hot key (did R increase by 100x, or per-request work increase by 100x, or….)? If R is unchanged, then the 10% of requests that hit the hot server will experience higher response times (roughly P90). But you need to know the per-server utilization to guess how much response time grows (proportional to the inverse of idle time).

(f) Give an example of an application that requires a "consistent assignment" as defined by Slicer.
Just name one we discussed, or make something up.

GroupServer from Lab #1. Or anything that requires at most one server for each key, e.g., write aggregation.

A few students interpreted elements of the scenario questions (Parts 2 and 3) narrowly. I have annotated the exam questions here with some remarks in bold. As always, partial credit was reasonably generous.

I was surprised by the number who were tripped up on 2(a). In addition to holding locks and pushing updates on commit, it is necessary to invalidate or update a key/value cached on a client cache no later than the point where it becomes stale, i.e., when some other client completes a write to that key.

Some students misread Part 4 and lost some points by answering (a) and (b) as questions about sloppy quorum systems. There was also some confusion on 4(c) and 4(d) about sloppy quorums. If Dynamo receives two concurrent writes for a key, it recognizes them as concurrent and returns both on read until the app writes a merged result. For 4(d) I gave credit for any concurrent write scenario, but I wanted you to say how it could happen: two coordinators, or a second write after reading a stale value.

For Part 5, some students proposed metrics that are useful but are hard to work with (like variants of Mean Squared Error) rather than the metrics discussed in class. A few stuck with it through the math. For these you got credit if it was clear from your answer that you understood how consistent hashing adapts to the scenarios.