CPS 512 midterm exam #1, 10/7/2016

Your name please: _____ NetID:_____

/60
/40
/10
/10
/20
/60
/200

Answer all questions. Please attempt to confine your answers to the boxes provided. If you don't know the answer to a question, then just say something else relevant for partial credit. T/Fs and shorts are 5 points each.

P1. Transactions: T/F (60 points)

Indicate whether each statement is true or false by placing a T or F to the **left** of the statement. You can place additional notes or conditions to the right.

1. In the two-phase commit protocol (2PC), a transaction T always commits if a majority of participants vote to commit T. *F: need unanimous vote.*
2. Commitment is durable in client-server transaction systems with 2PC, as long as a majority of participants survive and retain their state, even in the presence of a network partition. F: *ACID is durable, but only if no participant loses its state.*
3. When a transaction T commits with 2PC, there is always some point in the protocol at which all participants have voted, but no participant knows the outcome of T. *T*
4. When a transaction T commits with 2PC and two-phase locking (2PL), each participant releases locks held by T when it votes to commit T. *F: must wait for outcome before releasing (and exposing values).*
5. Failure of the coordinator during a two-phase commit (2PC) never forces the transaction to abort. *F: may abort on recovery, esp. if coordinator has state. I accepted "T" if you explained that coordinator can restart protocol when it recovers (thus never necessary to abort). This is not a good question, sorry.*
6. Failure of a participant other than the coordinator during a 2PC (failure of a storage server or RM) always forces the transaction to abort. *F: participant may fail after prepare/vote; cannot abort transaction.*
7. In client-server transaction systems, failure of the client during a 2PC always forces the transaction to abort. *F: client/coordinator can restart protocol.*
8. In a transactional server with a NO-STEAL/NO-FORCE buffer manager (e.g., Birrell/Wobber), the server always writes updates to the log before it responds to the client.
9. In a transactional server with a NO-STEAL/NO-FORCE buffer manager (e.g., Birrell/Wobber), the server does not write updates to the file system until after it has responded to the client.
10. In a transactional server with a NO-STEAL/NO-FORCE buffer manager (e.g., Birrell/Wobber), taking checkpoints (snapshots) more frequently increases the cost of recovery.
11. In a transactional server with logging, each update record in the log must identify the transaction that issued the update.
12. Any transactional service is linearizable if each request is executed at most once.

8. In a transactional server with a NO-STEAL/NO-FORCE buffer manager (e.g., Birrell/Wobber), the server always writes updates to the log before it responds to the client. *T: write-ahead-logging, and commit before replying.*

9. In a transactional server with a NO-STEAL/NO-FORCE buffer manager (e.g., Birrell/Wobber), the server does not write updates to the file system until after it has responded to the client. *T: NO-FORCE → don't write to home until after commit, reply immediately after committing. I accepted T if you explained that the server doesn't *have* to reply before writing or that the log write is likely a file system write.*

10. In a transactional server with a NO-STEAL/NO-FORCE buffer manager (e.g., Birrell/Wobber), taking checkpoints (snapshots) more frequently increases the cost of recovery. *F. It reduces the cost because we can truncate the log after snapshotting, so there are fewer log entries to inspect on recovery.*

11. In a transactional server with logging, each update record in the log must identify the transaction that issued the update. *T: I accepted F if you said it is unnecessary for single-threaded transactions (e.g., as in Birrell-Wobber), since they commit serially.*

12. Any transactional service is linearizable if each request is executed at most once. *T. Serializability of transactions implies that all ops on individual objects appear (to all clients) to execute in a serial order. Many answers said it has to be "exactly once" to be linearizable. But "exactly once" means "at most once and at least once". But linearizability does not require every request to execute at least once: if a request does not execute, it does not matter what order it did not execute in!*

P2. More T/Fs on Consensus and Consistency (40 points)

Indicate whether each statement is true or false by placing a T or F to the **left** of the statement. You can place additional notes or conditions to the right. **Consensus** refers to Paxos, Viewstamped Replication (VR), or Raft. These protocols are used for primary/backup replication, e.g., Replicated State Machine (RSM) replication.

1. In Consensus, any partition in the network always results in denial of service (no availability) on at least one side of the network partition. *T: can't have a majority on both sides of a partition, Consensus requires a connected majority in order to make progress.*
2. In Consensus, any partition in the network always results in denial of service (no availability) on at most one side of the network partition. *F: a partition may separate the network into >2 "sides".*
3. In Consensus, no participant (other than the primary) executes a request until all participants have voted on it. *F: an operation can commit if only a majority (including the primary) have voted on it. Replicas execute committed operations as soon as they learn of commitment.*
4. In Consensus an operation always commits if the majority of participants vote to commit it. *T: once a majority have voted, the recovery protocol ensures that commitment is stable. If the primary fails before learning of commitment, the new primary learns of the committed update and recommits it.*
5. Commitment is durable in Consensus replication as long as a majority of participants survive and retain their state, even in the presence of a network partition. *T: the participants are replicas, and any majority must include at least one replica that knows of the update. When the partition heals, all committed operations are recovered.*
6. A Chubby client can be in jeopardy only after it acquires a session lease at least once. *T*
7. Any service that is replicated with Consensus is linearizable if each request is executed at most once. *T. Consensus commits in a total order at all replicas; all requests serialize through the primary; the primary's reply to any client reflects the total order.*
8. Any service is linearizable if each request is executed at most once. *F. Services may have weak consistency. E.g., eventual consistency in Dynamo is not linearizable since a reader may receive stale data rather than the value of the last write.*

P3. Replicated State Machine (10 points)

Please confine your answers to the space provided.

1. The Replicated State Machine (RSM) model of primary-backup replication with Consensus does not scale: it does not yield higher throughput as we add servers. Why? What could go wrong if we permit any RSM replica to act as the coordinator for a write operation?

The primary is a bottleneck: the primary must execute all requests. More generally, every replica must execute every request, so adding replicas increases the total cost per request, and does not improve performance. The Consensus protocol also adds overhead, so it may have slightly slower throughput than a single unreplicated server.

One purpose of using a single primary is to sequence all requests into a total order. If we allow other replicas to act as coordinator for writes, we lose that property, sacrificing linearizability.

2. What could go wrong if we permit RSM backup servers to execute read operations?

A read could return stale data if it is served by a replica that has not yet learned of a committed write. And this is possible because a write may commit before all replicas have acknowledged it: only a majority is needed.

P4. Chubby (10 points)
Answer each question with a sentence or a short bullet list. Please confine your answers to the space provided.

1. Chubby is a coordination service that uses primary-backup replication with Consensus, and yet it claims to be scalable to large numbers of clients. How does it overcome the scaling limitation?

Caching. Other answers were also worth some points: in particular, session-grained leases with configurable lease interval. Some people said "sharding across multiple cells": that was worth something, but any coordination requires that clients are served by the same cell.

2. When a Chubby server receives a request to update an object/node, it may send messages to other servers in its cell and to other clients before completing the request. List three reasons for these messages.

My three: propagate the update to replicas in the cell, send cache invalidations to clients, send notifications to clients with watches on a node. There are others. Session renewal may be piggybacked opportunically on these messages, but does not count as a "reason" for the message.

P5. RAMcloud (20 points)
Answer each question with a sentence or a short bullet list. Please confine your answers to the space provided.

1. RAMcloud is a key-value store that uses primary-backup replication with Consensus, and yet it claims to be scalable to large (datacenter) server clusters. How does it overcome the scaling limitation?

Sharding! And caching. I marked a "minus" for not mentioning caching, but that was only a point or two. RAMCloud's use of DRAM is not pertinent here: it affects the throughput of each server, but does not impact scaling to a large cluster. Similarly, use of a fast network improves latency and maybe removes a limitation on traffic scaling, but doesn't address how to scale RAMCloud to a cluster.

2. RAMcloud uses RIFL, which provides stronger linearizability assurances than classic RPC (e.g., Birrell/Nelson) because completion records are durable. Sketch an example to show a scenario in which loss of a completion records can lead to a violation of linearizability in classic RPC.

This is given in the RAMcloud paper and on the slides. The server executes an RPC update request twice. But that is not enough: you really need an intervening client (or at least another update request) to observe a linearizability violation. Half credit for leaving it out.

3. RIFL has higher latency than classic RPC, because the completion record for each RPC must be made durable (e.g., by writing it to disk) before issuing a reply to the client. Yet the authors of RIFL claim that the latency cost is negligible for services that support fail-over. Why is the latency cost negligible for these services?

Each RIFL request pertains to an object. With fail-over, an update request to the object is replicated before replying to the request. The completion record is replicated with the update and stored with the object on the backups, so the "marginal" cost of RIFL is low given RIFL's assumption that the service is already paying this replication cost (e.g., using Consensus). Note that this question was about latency during normal operation, and not about the cost of fail-over or recovery.

P6. Sharding (20 points)

Given a scalable key-value store (like RAMcloud), any application-tier server (the clients of the K/V store) can access any of the service's data, so in principle any application server can handle any request. Yet many services apply a request routing function to route each request to a selected application server based on the data used by the request: sharding. Give two reasons why sharding can be helpful in the application tier in this setting.

1. *Caching is more effective if requests to each given application-level object are concentrated on a small spread of application-tier servers. Those servers handle requests for a smaller number of objects, so they have room to cache a larger share of them. Also, all requests to those objects hit the same server(s), so it is more likely that a previous request has already fetched data into the cache.*

2. *Consistency and concurrency control are easier. For example, if only one application server operates on any given key/value pair, there is no need to coordinate with other application-tier servers over that key/value pair.*

Some answers gave me generic reasons why sharding is good (load balance, scalability, etc.). But the question is about sharding can do better than (say) random distribution of requests, or round-robin, given that "in principle any application server can handle any request".

P7. Abstraction boundary (20 points)

In modern distributed services, the API that the application sees is typically different from the RPC API for the service. Why? Illustrate with examples from NFS, Chubby, RAMcloud/RIFL, and other services we have discussed.

This is a question about the client library, which implements the "API that the application sees", and issues requests to servers using the "RPC API for the service". I wanted examples from at least these three systems. NFS: caching, lease-based consistency, asynchronous writes, single-component lookup. Chubby: caching and lease-based consistency, jeopardy notifications. RAMCloud/RIFL: sharding, two-phase commit, commit recovery. These features reside in the client library for good reasons. I gave full credit for answers that gave me a couple of good examples (2/3) and explained them well enough.

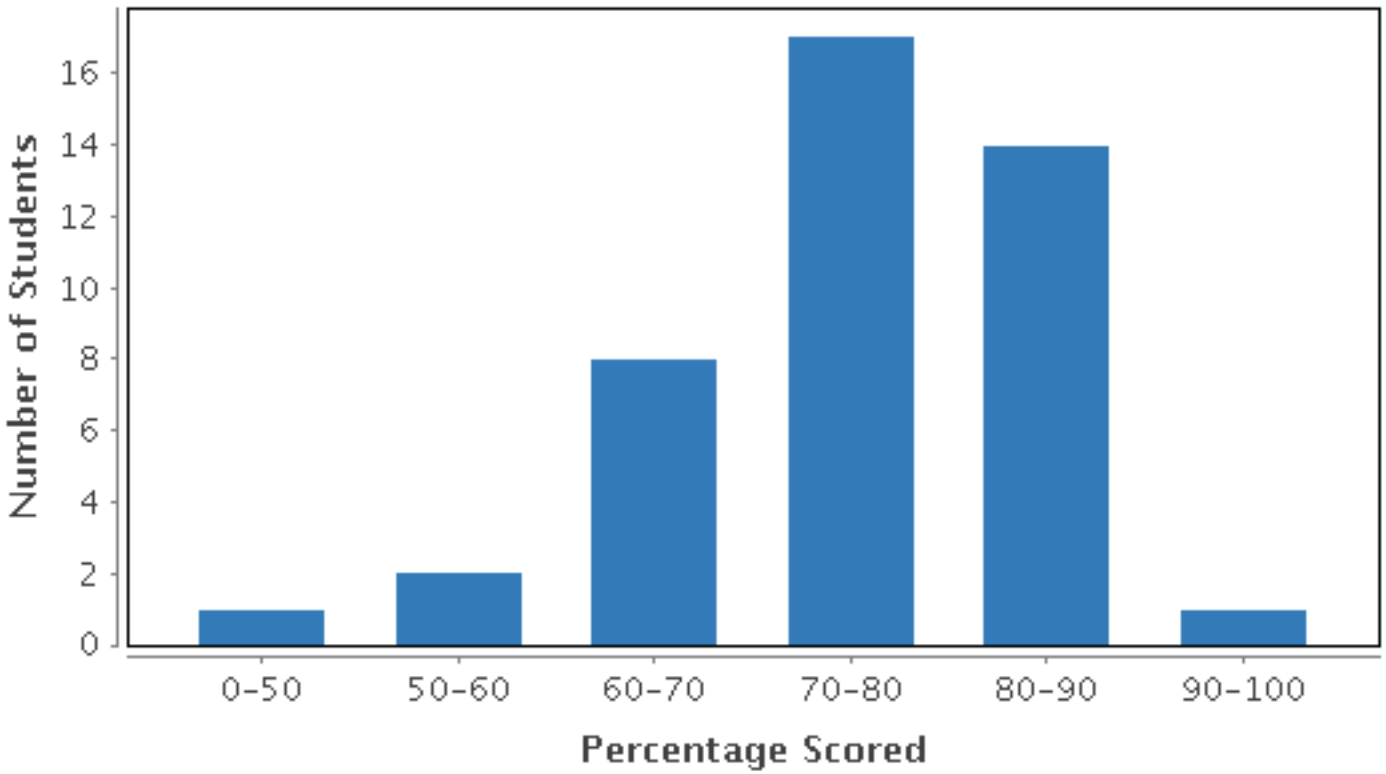
A few answers talked about RPCs used inside the service (e.g., for replication and primary selection), or features at the RPC protocol level (below either API). These are tangential to the point of the question, but were worth something.

P8. Serializability vs. Concurrency (20 points)

Transactional serializability does not imply serial execution. Give a specific example of two transactions for which any interleaving of operations is a serializable execution.

I accepted answers in which all operations were reads, or in which T1 and T2 have no data items in common. I also accepted answers with a single blind-write conflict, but I was less happy about those.

Grade Statistics for Midterm #1



Average (Mean) Score	146.51 / 200 (73.26%)
Median Score	150 / 200 (75%)
Standard Deviation	27.96
Lowest Score	0 / 200 (0%)
Highest Score	188 / 200 (94%)
Total Graded Scores	43