/40
/50
/50
/60
/200

CPS 512/590 first midterm exam, 10/6/2015

Your name please: _____

Part 1. Managing replicated server groups

These questions pertain to managing server groups with replication, as in e.g., Chubby, Dynamo, and the classical Replicated State Machine (RSM) model introduced in the 1978 Lamport paper *Time, Clocks, and the Ordering of Events*…. Answer each question with a few phrases or maybe a sentence or two. [40 points]

a) *Consistent Hashing* algorithms select server replicas to handle each object (e.g., each key/value pair). They use hash functions to assign each object and each server a value within a range referred to as a "ring" or "unit circle". In what sense is the range a "ring" or "circle", and why is it useful/necessary to view it this way?

The range is a "ring" because the **lowest value in the range is treated as the immediate successor of the highest value in the range for the purpose of finding the replicas for an object**. The N replicas for an object are the N servers whose values are the nearest successors of the object's value on the ring. If there are not N servers whose values are higher than the object's value, then the **search wraps around** from the end of the range (the highest value) to the start of the range (the lowest value).

b) In practice, it is important for consistent hashing systems to assign each server multiple values for multiple points on the ring. Why?

If a server leaves the system, its immediate successor(s) on the ring take responsibility for its objects. By giving each server multiple values (tokens or virtual servers), the **server's load is spread more evenly among many servers if it leaves**. Similarly, if a new server is assigned multiple values on the ring, it draws load from more servers. Another benefit is that a server's load is proportional to the number of tokens it has, so it is easy to weight each server's load to match its capacity.

c) In the classical RSM model, it is important for all replicas to apply all operations in the same order. How do Chubby, Dynamo, and the Lamport RSM system achieve this safety property during **failure-free operation**?

Chubby and Dynamo: primary/coordinator assigns a sequence number for each update; replicas apply updates in sequence number order. In Chubby the primary is always unique and assigns a total order to the updates. Dynamo may fall back to a partial order using vector clocks if the primary/coordinator is not unique (generally because of a failure). Lamport RSM: operations are **stamped with the requester's logical clock**. If two ops have the same stamp, their order is determined by their senders' unique IDs. Thus the order is total.

CPS 512/590 first midterm exam, 10/6/2015

Part 1. Managing replicated server groups…

d) The Lamport RSM system is fully asynchronous and does not handle failures of any kind. Even so, it requires each peer to acknowledge every operation back to its sender. Why is this necessary, i.e., what could go wrong if acknowledgements are not received?

The acknowledgements **inform the ack receiver of the ack sender's logical clock**. Since each node receives messages from a given peer in the order they are sent (program order), each ack allows the ack receiver to know that it has received any operations sent with an earlier timestamp from the ack sender. If acks from a peer are lost, the other **peers cannot commit any operations**, because they cannot know if the peer has sent an earlier request that is still in the network.

Part 2. Leases

These questions pertain to leases and leased locks, as used in Chubby, Lab #2, and distributed file systems like NFSv4. Answer each question with a few phrases or maybe a sentence or two. [50 points]

a) In Chubby (and in Lab #2), applications acquire and release locks with explicit **acquire()** and **release()** primitives. Leases create the possibility that an application thread holding a lock (the lock holder) loses the lock before releasing it. Under what conditions can this occur?

A **network partition**. That is the only case! (OK, it could also happen if the lock service fails.) Otherwise, the client renews the lease as needed until the application releases the lock. The lock server does not deny a lease renewal request from a client, even if another client is contending for the lock.

For this question it was not enough to say "the lease expires before the application releases the lock", because a correct client renews the lease in that scenario, and the lease is lost only if some failure prevents the renewal from succeeding.

CPS 512/590 first midterm exam, 10/6/2015

Part 2. Leases…

b) In this circumstance (a), how does the lock client determine that the lease has been lost? How does the application determine that the lock has been lost? Be sure to answer for both Chubby and your Lab#2 code.

For both, the lock client knows the lease is lost if the lease expires, i.e., the client receives no renewal response before the expiration time.

Chubby: the application receives an exception/notification that it is in **jeopardy** and (later) that it has lost its session with Chubby.

Lab #2: I accepted various answers. You were not required to handle this case. In class, I suggested that you renew leases before they expire (by some delta less than the lock hold time in the application), and do not grant an acquire while a renewal is pending. That avoids this difficult case. Note that it is not sufficient to notify the application with a message: for actors, the notification message languishes at the back of the message while the actor's thread is busily and happily computing, believing that it still holds the lock.

c) Chubby maintains a single lease for each client's session with Chubby, rather than a lease for each individual lock that a client might hold. What advantages/benefits does this choice offer for Chubby and its clients?

It is more **efficient**: the cost of handling lease renewals is amortized across all locks held by a client. It is also more tolerant to transient network failures and server fail-over delays because Chubby renews a client's session lease automatically on every message received from a client. If the session lease is lost, then the client loses all locks and all other Chubby resources that it holds.

d) Network file systems including NFSv4 use leased locks to maintain consistency of file caches on the clients. In these systems, what events cause a client to request to acquire a lock from the server?

Read and write requests (system calls) by the application to operate on files.

e) In network file systems the server can recall (callback) a lock from a client, to force the client to release the lock for use by another client. What actions must a client take on its file cache before releasing the lock?

If it is a write lease: **push any pending** writes to the server. Note that this case differs from (a) and (b) in that the NFS client is itself the application that acquires/releases locks, and it can release on demand if the server recalls the lease.

CPS 512/590 first midterm exam, 10/6/2015

Part 3. Handling writes

These questions pertain to committing/completing writes in distributed storage systems, including Chubby and Dynamo. Answer each question with a few phrases or maybe a sentence or two. [50 points]

a) Classical quorum replication systems like Chubby can commit a write only after agreement from at least a majority of servers. Why is it necessary for a majority of servers to agree to the write?

Generally, majority quorums ensure that **any two quorums intersect** in at least one node that has seen any given write. In Chubby, majority write is part of a consensus protocol that ensures that the primary knows about all writes even after a fail-over: a majority must vote to elect the new primary, and that majority set must include at least one node that saw any given write.

b) Dynamo allows writes on a given key to commit after agreement from less than a majority of servers (i.e., replicas for the key). What advantages/benefits does this choice offer for Dynamo and its clients?

Lower latency and better availability: it is not necessary to wait for responses from the slower servers, so writes complete quickly even if a majority of servers are slow or unreachable.

c) Given that Chubby and Dynamo can complete a write after agreement from a subset of replicas, how do the other replicas learn of the write under normal failure-free operation?

They are the same: the primary/coordinator sends (multicasts) the write to all replicas. Although they do not wait for all responses, **all replicas receive the update from the primary/coordinator** if there are no failures.

d) What happens in these systems if a read operation retrieves a value from a server that has not yet learned of a recently completed write? Summarize the behavior for Chubby and Dynamo in one sentence each.

Generally, quorum systems read from multiple replicas: at least one of these replicas was also in the last write set (R+W>N), and so **some up-to-date replica returns a fresh value and supersedes any stale data read from other replicas**. Dynamo behaves similarly, but it is possible that the read set does not overlap the last write set (R+W<N+1), and in this case alone a **Dynamo read may return stale data**. For Chubby, this was unintentionally a trick question: the stale read case “cannot happen” because Chubby serves all reads from the primary. (Chubby’s consensus protocol pushes all pending writes to the new primary on fail-over). I accepted a wide range of reasonable answers for Chubby.

CPS 512/590 first midterm exam, 10/6/2015

Part 3. Handling writes…

e) Classical ACID transaction systems differ from Dynamo and Chubby in that they require agreement from *every* server involved in a transaction in order to commit (e.g., as in the two-phase commit protocol). Why?

Atomicity (A) requires that **each server that stores any data accessed by the transaction must know its commit order** relative to other transactions. This is necessary to ensure that all servers apply data updates in the same (serializable) commit order.

Each server must also know whether/when the transaction commits or aborts so that it may log updates to persistent storage (on commit) or roll them back (on abort), and release any locks held by the transaction. This is necessary for the "ID" properties.

Part 4. CAP and all that

The now-famous "CAP theorem" deals with tradeoffs between Consistency and Availability in practical distributed systems. Answer these questions about CAP. Please keep your answers tight. [60 points]

a) Eric Brewer's paper *CAP Twelve Years Later* summarizes the CAP theorem this way: "The CAP theorem asserts that any networked shared-data system can have only two of three desirable properties." It then goes on to say that "2 of 3 is misleading", citing a blog by Coda Hale that suggests a system cannot truly be "CA", and that the only real choice is between "CP" and "AP". Do you agree? Explain.

A good argument is: any system could be subject to network partitions. The question is, how does the system behave if a partition occurs and (as a result) some server cannot reach some peer that it contacts to help serve some request? One option is to block or fail the request, which sacrifices availability (A). Another option is to go ahead and serve the request anyway, which sacrifices consistency (C). There are no other options. In other words, you can't choose not to be "partition-tolerant" (P). If a partition occurs your application will lose either C or A: the question is which.

CPS 512/590 first midterm exam, 10/6/2015

Part 4. CAP and all that…

b) Modern abstractions for building distributed systems expose failures to applications to varying degrees, so that they may manage their own C/A tradeoffs, or at least compensate for failures that occur. Illustrate with reference to Chubby and Dynamo.

Consider a network partition.

In Chubby, the minority side of the partition cannot make progress, because writes require majority acceptance. Chubby exposes failures to application servers (on the minority side) by generating an event to notify the application that it is "in jeopardy" or that it has lost its session with Chubby, and so all handles it held are invalid.

In contrast, Dynamo would accept writes on both sides of a partition and then rely on the application to fix up any inconsistencies after the partition heals. Writes accepted on the minority side may conflict with those accepted on the majority side. After the partition heals, Dynamo exposes the failure to the application by notifying it of any conflicting writes that Dynamo accepted and that the application must reconcile to restore consistency.

c) CAP was originally motivated by an argument that ACID transactions (e.g., as implemented with classical two-phase commit) do not enable applications to manage tradeoffs of consistency and availability during failures. Do you agree? Explain.

A good argument is: by definition, ACID transactions do not accept any loss of consistency without violating the ACID properties. All servers participating in a transaction must agree to the commit order: if this is not possible due to a failure, then transactions cannot commit, sacrificing availability. In particular, common protocols like two-phase commit have unbounded hanging/blocking states for certain failures. However, implementing highly available transactions is still an open research problem.