Name: AndrewID:

15-640/440 Homework #2
Fall 2013 (Kesden)

Replication and Quorums

1. Consider replication to multiple servers based upon a write-one, read-all static quorum with version numbering. What steps must be taken upon a write to ensure the correctness of the version number?

   The initial version number needs to be read from at least a read quorum of servers. If we take this initial version number, bump it by one, and use the result as the new version number for the update, we run the risk that another update has already done the same, yielding two different updates with the same version number and no detected conflict. If we, by atomic operation, obtain the version number and write-lock from a read-quorum, we can then safely bump the version number, complete the write, and release the lock, without the risk of having a version number shared by different objects. (Of course, we could still lose an update, if someone else writes after this updated based upon a version read before this update).

   The only other option might be to use physical time stamps – and be willing to accept the attendant risk associated with the window of time by which physical clocks might differ. This is not unreasonable, for example, in a situation such as a single user's home directory, etc, where all uses within a small interval likely originate on the same machine and many conflicts might be when the user switches machines after an extended period away, e.g. after going home for the day.

2. Consider Coda Version Vectors (CVVs). Give an example of two concurrent CVVs. What is indicated by concurrent CVVs? Please describe one situation in which this might occur.

   <1,1,2,2> and <2,2,1,1>

   Concurrent version numbers indicate that updates happened in different partitions, during some type of partitioning. Perhaps it was a true partitioning, or perhaps there are weakly connected clients choosing to interact with only a subset of servers. But, in any case, independent updates occurred, each to a different subset. The problem indicated is that there are conflicting versions.

3. Consider a situation with 5 replica servers employing a write-2/read-4 policy and version numbering. Please design and describe a locking scheme that ensures that version numbers remain correct, even in light of concurrent writes. You do not need to consider failure. But, you do need to describe all necessary communication and state, such as communication between or among servers and/or participants.

   See question #1 for a discussion of the necessary locking.

4. Please consider the special case of a mobile client using a read-one/write-all quorum. Please design an efficient locking protocol to ensure concurrency control. But, please ensure that your protocol permits the lock to be acquired at one replica and released by another.

   The discussion here is essentially in question #1, except now we may want to have the client communicate with only one server, which acts as its proxy and communicates with the rest. This minimizes the use of the "straw" by the weakly connected mobile client and decreases the time locks are held, improving the throughput of the whole system.

5. Coda Version Vectors (CVVs) are a form of logical time stamp, specifically a vector time stamp, which are used to aid in the management of replication. What could cause two CVVs be *concurrent* (and non-identical)? Why? Illustrate your answer with an example involving 2 servers and 2 clients.

   Duplicate question. Sorry. See question #2.

Agreement, and the Impossibility thereof

6. Consider content servers located at three distant points across the globe. Can we devise a system to ensure that, at any point in time, *exactly* one of them is visible to each client? Please assume that no more than two are unavailable at any point in time, and that it is possible for more than one to be available to provide server at any point in time. Why or why not? *Hint: Consider the network, all its pieces, and all its failure modes.*

   Nope. Not possible. A client or server can't tell if another server is down or just not accessible, such as due to a network partitioning. Servers certainly can't tell what might or might not be visible to clients that might be on the other side of a partition.

7. Assume that we have two magical networks. The first, a *Worm Hole*, truly has zero latency. But, unfortunately, each time a message is sent, it is lost with some probability *p*. The second, *Perfect Fiber*, has a latency directly attributable to the speed of light, but never loses or corrupts a message. Reliable protocols, such as TCP, are available for use on either network. Which network has a faster *guaranteed* delivery time? Why?

   The key word here is guaranteed. A perfect network can offer a guarantee. The rest we can do with any other type of network is keep rolling the dice.

Recovery: Checkpointing and Logging

8. Consider checkpointing as means of preparing for failure in a large scale distributed system. What is the most significant challenge that arises when attempting to recover the state of a distributed computation from checkpoints or log entries recorded only at individual participants? Why is preventing this problem, by its nature, very challenging?

   Coordination across participants is very challenging. If one system recovers to a checkpoint that is even slightly old, it might be inconsistent with the rest of the systems state, for example, it might contain stale values. This can force costly cascading rollbacks.

9. Consider logging as a means of preparing for failure in a large scale distributed system. Why do recovery facilities rarely rely upon logging, without any checkpointing, for recovery?

   Playing back logs can take a long time. If an entry is ever lost along the way – the state can't be restored completely and the amount of spoilage may be hard to determine.

10. Most of the time, for the purpose of recovery, logging of messages is performed at the recipient. Should it become necessary, this enables the host to replay the logged messages to recover its state. Imagine for a moment that the only log of messages exists only the senders, perhaps because the recipient is a thin client without significant storage. Can the thin client recovery its state simply by asking the senders to replay the messages they sent? If so, what property of the replay ensures that the recipient will be restored to the pre-crash state? If not, what is the problem and how can it be addressed?

    The problem with receiving uncoordinated replays from various senders is that they may, upon replay, interleave differently than they did originally. It can be addressed, as described in the notes, by having the recipient provide the sequence number to the sender as part of its reply. This will enable the sender to log the message with the receiver-side sequence number, and replay it with the number, proving a correct numbering to order the replay of messages from multiple senders.