Solutions: Assignment 3

1a) Much like in assignment 2, we are partitioning the dataspace such that only a subset of the nodes in the system is responsible for each piece of data. Note that, in this assignment, we assume the number of replicas in the quorum is based on the write quorum size rather than the total number of nodes in the system. In order for this to be true, we assume there is some background task that invalidates old replicas (you don't need to state this assumption in your answer).

In order to satisfy the requirement of $N_w = 2$, we would need to partition the space either into 4 quorum groups where N = 3 for each group, or 6 quorum groups where N = 2 for each group. You should describe how the dataspace is partitioned into these groups. Any mention of either hashing, or lexicographical ordering would be sufficient.

1b) With a highly skewed popularity distribution, the partition that is hosting the most popular data item is likely to have significantly more load than the other partitions. This load imbalance limits the request rate that the system can handle.

2a) Rather than use partitioning, we can limit the number of replicas by using erasure coding in our quorum. Split each data item into 5 fragments, use erasure coding to generate 10 fragments, and distribute a fragment to each node in the write quorum. This will limit the replication factor to 2. In order to ensure that the read quorum can reconstruct the original data, the read quorum size must be 7, which accounts for the required 5 fragments in addition to the 2 nodes that might not have the most recent version of the data.

2b) 5 nodes can fail before this replication scheme loses data. This is because the original data can still be reconstructed from fragments on the remaining nodes.

2c) By not partitioning the dataspace, we no longer face a load imbalance problem. However, rather than having a read quorum size of 2 or 1, we now require a read quorum size of 7, which introduces a fixed amount of additional load per query.

To simplify the analysis, let's only look at the load introduced by reads to the most popular data item. Assume that each node can handle a request rate of 100 requests per second. Using erasure coding, each node only participates in 7/12 of the requests. Therefore, the maximum number of requests per second the system can handle is 12 (100) / 7 = 171.43 requests per second. Compare to the partitioning schemes with 2 or 3 nodes per quorum, the maximum request rates they can handle are 200 and 150 request per second respectively. Therefore, it provides read throughput that is in between the two different quorum configurations for this type of skewed popularity distribution.

3) There are many ways to provide fairly efficient implementations of the different client-centric consistency models. Generally, we should have a notion of sessions consisting of the read and write operations that a client issues within some time interval, where the interval is determined by the client and the client is not concerned about the consistency of operations between intervals.

Servers should assign vector timestamps to objects on writes in order to determine when they arrived, and an object's vector timestamp should accompany it on reads, which include reads pertaining to disseminating replicas.

Clients should include the previous vector timestamp of the objects on read/write requests to the servers; it may be helpful to also specify which server the previous version came from. The server may then need to fetch items in responses to these requests depending on the client-centric consistency model that it is trying to provide.

The textbook (section 7.5.5) offers an alternate approach where nodes assign a timestamp to each write operation, and keeps track of a vector timestamp that represents the timestamp of the last write operation received from each of the other servers. Clients keep track of a vector timestamp representing the latest timestamp that it has received from each server, and this timestamp is sent to the server on reads and writes to help it determine which server it needs to fetch fresh data from in order to maintain client-centric consistency. This approach is less fine grain but reduces the number of timestamps that are needed (per node rather than per object).

These replication protocols typically assume there is some background replication process, perhaps based on an epidemic protocol that slowly propagates the data between the servers in the absence of additional client requests.

a) For Monotonic reads, the server should ensure that a previous object's vector timestamp is less than or equal to the vector timestamp of the object it is returning. Otherwise, fetch a newer version from one of the other servers.
b) For Monotonic writes, the client must include in its write request the vector timestamp that it received in response to a previous write request for the same object. The server will then ensure that it integrates that previous write request (by fetching a sufficiently new version of the object from one of the other servers) before accepting the new write request.
c) Similar to b, but also apply this on read requests.
d) Client should include in its write requests the latest vector timestamp it has received in response to a read operation for that object. The server must then ensure that it updates its data store with a version of the object that is at least as new as the timestamp it received before processing the write request.

4a) W1(x=3)R1(y=5)R2(x=3)R3(x=3)R3(y=5)R3(z=6)W2(x=2)W2(y=1)R2(z=6)

4b) W1(x=3)R3(X=3)R1(y=5)R3(y=5)R3(z=6)R2(x=3)W2(x=2)W2(y=1)R2(z=6)

Note: Transaction 2 aborts and must be restarted. I kept the transaction 2 label for the restarted transaction, but you can also call it transaction 4.

4c) Same as 4b

5a) There are many correct answers to this question. For primary copy remote-write, you should mention that the primary copy is stored at the CCCN, and that the CCCN is responsible for keeping

track of its replicas. If a node leaves the neighbor set of the primary node, the primary should replicate its objects to its new neighbor. Nodes should periodically check, for each replica it is holding, whether or not the primary is still in its neighbor set. If it is not, it should remove the replica. Backup nodes can handle read requests in this scheme. When responding on behalf of the primary, a node should check whether or not the primary is still in its neighbor set. If not, it should forward the request to a node that is still in the CCCN's neighbor set or to the CCCN, and then remove the replica. On primary node failure, the new CCCN needs to take over as the primary. This is fairly easy to detect on write requests, as the write request is forwarded to the new CCCN which then serves as the new primary. It can also be detected during standard ring maintenance.

5b) In a quorum-based scheme, read and writes complete once they are sent to any three nodes in the union of {CCCN} and the CCCN's neighbor set. One approach is to route to the CCCN and if it happens to go through any of the CCCN's neighbor set along the way, then include them in the read/write set. Once it arrives at the CCCN, fetch additional nodes from the CCCN's neighbor set to complete the read/write set. However, this somewhat defeats the purpose of using a quorum, as the CCCN still serves a special purpose and is a part of every quorum.

A slightly different approach is to have the client randomly pick 3 out of the 5 possible nodes before it sends it requests, and then modify the routing protocol such that it is sent directly to those 3 nodes without necessarily having to go through the CCCN. For example, the client specifies ahead of time that a particular request should be sent to node {1, 2, 5}, where node 3 is the CCCN. Nodes would keep track of its replica "position" for each key, and upon receiving its request, it would route to the next nodes in the set.

Handling node joins and leaves become a little bit trickier. Again, rather than relying on a primary node to ensure the number of replicas for each key is maintained, each node can instead use an epidemic protocol between the nodes in its neighbor set to determine if it has the latest version, and if it does, count how many replicas remain in the system. If the number of replicas is less than 3, it can choose another node to replicate to. This can lead to having more than 3 replicas, which is generally okay. One possible approach to delete excess replicas is to delete it from the node with the smallest or largest ID, or use any deterministic technique based on the hash of the key. A node with a replica that is no longer in the neighbor set of the CCCN can also initiate migration of its replica to a node that is in the CCCN's neighbor set.