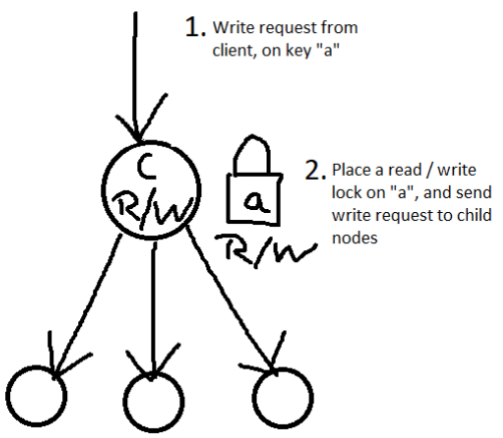
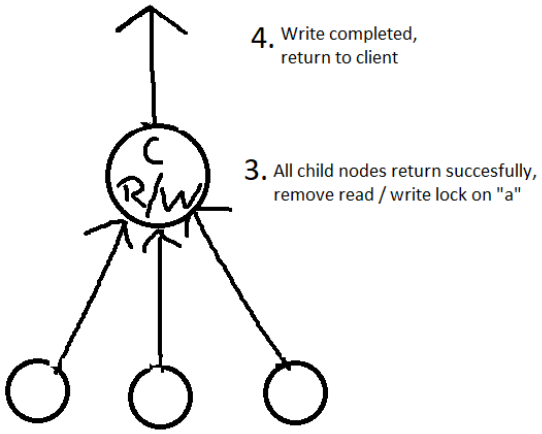
Rubal Goyal, Brayden Thompson
CS 455 Project 3
April 14, 2023

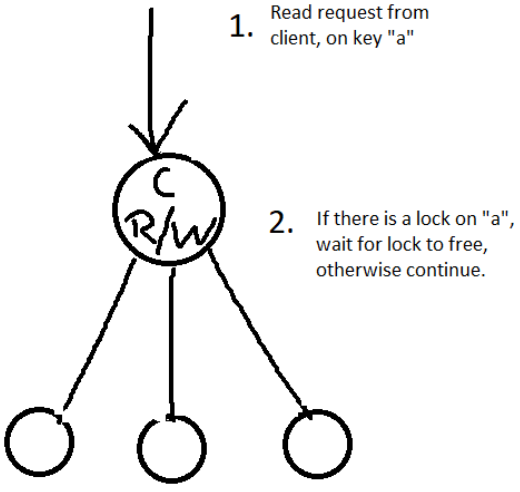
General Architecture:
Our server architecture will be attempting to implement a strict consistency model, and prioritizing the speed of reads over writes. In order to do this, we plan on having all client requests go through a single coordinator node. During writes, this node will place a read and write lock on the piece of data being written, preventing other clients from accessing it until the write has completed on all child nodes. After the coordinator has received confirmation of the write from all of the child nodes, it will return to the client, and the lock will be removed. For a read operation, there can be multiple read requests in progress at the same time, just not while a write is occurring.

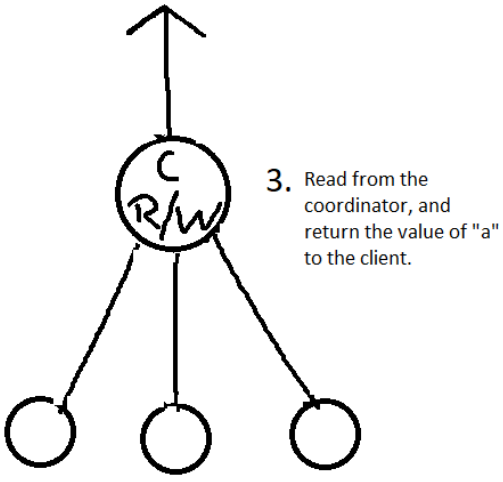
Performing a write:





Performing a read:





Consistency model and guarantees:
Since we are prioritizing the consistency of the data over availability, with this model we are guaranteeing that all of the data on each of the server replicas is the same after a write completes. From the clients perspective, writes are performed synchronously, and the coordinator is sending the write to the child nodes in parallel. By doing this, we can guarantee most of the client-centric consistency models, such as eventual consistency, read your writes, writes follow reads, and monotonic writes. Additionally, since we are also blocking reads while a write is in progress, we can also guarantee monotonic reads.

Fault tolerance:
With this model, we also have a good level of fault tolerance. If any of the child nodes go down, once they are recognized as being down and decommissioned, there will be no impact on the consistency of the data, since every child node has the same data. If the coordinator node goes down, there will be a period of time when the server cluster is inaccessible while a new coordinator is chosen, but this would not affect our consistency model. If the coordinator goes down while a write is in progress, there could be some degree of data lost depending on if the write went through on any of the child nodes. We go more into detail about this in the faults section.

Performance:
As for performance, the throughput of the server cluster is not great, since all requests from clients have to go through a single coordinator node. Additionally, writes will be quite slow, since the write has to successfully complete on every child node in the cluster before the coordinator can return. This has the benefit of good read performance, as assuming that there is no write in progress, the coordinator can fetch the data from itself, and quickly return. Otherwise, the read must wait for the write to complete.

Scalability:
This system will also be easily scalable as far as the number of nodes, as a new node just has to find out what node is the coordinator, and receive the most recent version of the data in order to be functioning. The throughput of the system does not scale at all as a result of the strict consistency requiring all traffic to go through the coordinator. The throughput will always be constant, no matter how many nodes are in the system.

Coordinator behavior:
Our communication model has a single coordinator node that manages all of the request traffic that the cluster receives. The coordinator will be chosen through an election process, likely by performing a Bully Algorithm. The coordinator node will be the single point of contact for any client requests, meaning that it will have a view on what requests are being processed in the server cluster at any given moment. Because the coordinator has this view, it will be able to enforce the fact that every child node will have the exact same data. This allows for strict client centric consistency, with monotonic reads and writes, since during a write operation, no other writes will be started, and no reads will be started. It allows for read your writes and writes follow reads, since a read following a write will not be started until the write has completely finished on

every child node. While a write operation is being performed, all other operations requested will be stored in a queue, and will be executed in that order. Additionally, every server has a list of active servers, and knows which server is the coordinator.

Fault Procedures:

- Child node crashes
    - Coordinator times out child connection, removes from server set
    - Contacts every child, telling them to remove the crashed node from server set
- Child node loses connection
    - The rest of the cluster:
        - Coordinator times out child connection, removes from server set
        - Contacts every child, telling them to remove the crashed node from server set
    - Child node:
        - Does not receive ping from coordinator, assume it is dead
        - Hold election, it becomes the new coordinator in a one server cluster
- Child node reconnects
    - Child node contacts coordinator, tells it to add itself to server set
    - Coordinator contacts every child, telling them to add new node to server set
    - Coordinator sends new node the most recent copy of the data
- Coordinator crashes
    - Child nodes do not receive ping from coordinator, assume it is dead, cluster is down
    - Child nodes send out election message, and bully algorithm starts
    - Once coordinator chosen, client can connect to the new coordinator
- Coordinator loses connection
    - The rest of the cluster:
        - Child nodes do not receive ping from coordinator, assume it is dead, cluster is down
        - Child nodes send out election message, and bully algorithm starts
        - Once coordinator chosen, client can connect to the new coordinator
    - The old coordinator
        - Coordinator times out every child connection, and removes them from the server set
        - Clients may or may not still be able to connect, if so, the old coordinator acts as a separate single node cluster until it reconnects to the old cluster.
- Coordinator reconnects
    - Connect to other nodes in the system,
    - Find out who is the coordinator if a new node enters the cluster. We can do this in two ways :
      Each node has a unique ID, and the node with the highest ID is the coordinator. When a node joins the system, it sends a message to all other nodes requesting the current coordinator's ID. If no response is received, the new node assumes

that it has the highest ID and becomes the new coordinator. If a response is received from a node with a higher ID, the new node defers to that node and becomes a follower.

Another approach is to use such as a DNS server to maintain a list of active nodes in the system and their roles. When a new node joins the system, it queries the service discovery mechanism to obtain the current coordinator's IP address or hostname, and then establishes a connection to the coordinator to participate in the system.

- Coordinator disconnects / crashes mid-write
    - Lots of different ways to handle this
    - One is to update a version number in each child node for each key, and when a new coordinator is chosen, the coordinator distributes the data with the highest version number to those with lower numbers.
        - In this case, after a crash, there will at most be a difference of one update between nodes for any given key, so if at least one child node successfully completed the write for that key, the data will be recovered.
    - Or implement the “two-phase commit” idea talked about in class, where a node knows that it has out of date data for a given key.