Design Document for C+P key value store
Assignment 3
CMPS 128

Cruz IDs: tmathias, jscalett, sborland, vispatel
Names: Theoren Mathias, Joyce Scalettar, Sarah Borland, Viraj Patel

Goal: The goal of this assignment is to create a key value store that fulfills either C+P or A+P. The P is mandatory, regardless of the strategy we employ. We must employ fault tolerance through a replication scheme of our choice.

Assumptions: We are assuming that handling Omission and Byzantine failures will be out of our scope, and that there will be 5 nodes brought up to start, and nodes may be partitioned/removed (though not added) at the grader's discretion. These nodes will know about every other node to start. We assume that the nodes can be partitioned off from each other at any time and the program must adapt accordingly.

Design: Our implementation will be to use primary backup with C+P. Each node will have a linked list containing information about the other nodes in the system. This list will keep track of each node's port number, node status (whether the node thinks that particular node is alive or dead), and node role (if the node thinks that particular node is the leader or backup) of every node including itself, as well as the port number of who they think the leader is, and the list queue of PUT/DELETE requests. When each node is brought up, it will elect itself the leader and set the flag/port to themselves as leader and then check to see if there is already a leader by pinging everyone for their information. If there is, the node will set itself as a backup and set the leader role flag and port in the node list for who they think is the leader.

The queue is a Python-native dictionary will consist of all the PUT/DELETE to-be-completed requests for the client. This queue will be managed by the leader and requests will be dequeued when the leader receives ACKs from the backups that they have all completed the request (See Leader/Backup Protocols below). All requests should have timestamps of when the leader added it to their queue. Each node will still have a copy of the queue (sent by the leader) so that the requests are not lost if the leader node goes down since they can become the leader if that happens.

Our leader election protocol will select the node that is alive with the largest port number and elect that node to be the leader. We decided on this because even though it's not fair, the ordering will be able to bring consensus among all nodes when it comes to who's the leader. This leader will then be sent the queues that each backup node was holding in case the leader crashed, and the new leader will use dictionary commands to merge them so that duplicate requests are discarded (in case the leader's last queue was only successfully sent to a few nodes in the network). This updated request queue will then be sent to all of the backup nodes during the ping and the new leader will start following leader protocols.

For our heartbeat ping, our nodes have an app route called */ack* where they return a response with their node's information in a form of a JSON file. When a node wants to ping a node, it will call a GET request to the node's */ack* route. If the request is successful, the */ack*'d node's retrieved information will be updated in the requester's node linked list else if the request failed, the /ack'd nodes status is updated to be dead.This heartbeat will start a leader election if no living node is detected as being the leader. This means that when a node goes down or the system is partitioned, the program will cope by electing a new leader to manage the system (see Leader Election below). Also, at each ping, it will check if the /ack'd node's port is higher than the leader's, if it is, it will set the /ack'd node as the leader, else if the /ack'd node is a leader and the requester's port is greater than the /ack'd port, the requester sets the */ack*'d node as a backup in its node list and flags itself as leader and begins following leader protocol. There is also the special case of multiple leaders; if the node getting the ping is the leader (pingee), the node that sent the ping also believes they are the leader (pinger), and the pinger's port number is smaller than the pingee's port number, then the pingee is the true leader now and will merge the pinger's queue with its own.

LEADER PROTOCOL

When receiving a PUT/DELETE request, the leader will place the request into its own queue and send the queue to all backups. At each heartbeat, it will then dequeue the earliest request in the queue and perform the request. It will then send the request to all alive nodes to perform with a timeout of .15 second. If one of the nodes timeout, the leader will ping it to see if it's alive. If it's still alive, it will try the request again else if it sees that the node is "dead" (or even behind a partition), it will ignore it while it waits for the rest of the alive node's ACKS. Once the leader receives ACKS from all nodes that the queued request has been performed, it will dequeue the request.

When a READ request is received, the leader will send the requested data to the client and await a response. On an OK response, it will do nothing. On a failed response, it will resend the data.

When a node becomes a leader through leader election, it will check with other nodes to ensure that its request queue is up to date, and then send out the most current request queue to the backup nodes for them to replace.

BACKUP PROTOCOL

When receiving a PUT/DELETE request from the leader, it will do the request and send an ACK to the leader. If receiving a queue from the leader, it will replace its queue with the leader's queue. If the request is from the client, the backup will forward the request to the leader for the leader to handle. If the backup receives a READ request from the client, it will send the data directly back to the client.

LEADER ELECTION

When at the start of election, the node sets the leader field in its node in the list to blank as a signal to the other nodes that they don't know who the leader is. They will then go through it's

list of nodes and find the node with the largest port and set that node as the leader (which could be itself.) It will then flag the chosen node as leader in the leader field and then on the next ping, it will send this information through it's */ack* route so that the other nodes can grab that information.

Reasoning: Our group decided that a C+P system was preferable for this project. Our philosophy with our implementation was "Drop the client, not the data," where even though our implementation may refuse/reject client, our data would be consistent across all nodes. We decided to use Primary Backup as our implementation because it has the functionality that we needed and made sense to all of us from a conceptual point of view. With Primary Backup, our system will be able to cope with nodes going down (using everyone's request queues being consistent with the leader's), and with the system being partitioned (both leader's request queues are merged with the larger port merging the smaller port's queue into its queue). Since all writes have to go through the leader, we know that no data would be written to all the nodes unless it goes through the leader and the leader confirms that all the nodes have written it. This will let the data that is stored or going to be stored stay safe and consistent.

Expectations: With our leader election and heartbeat implementations, we expect our program to run fairly well under a variety of circumstances. If a node goes down, the program will realize it through the heartbeat and adjust accordingly either by holding a leader election or merely noting it in it's linked list of node information. If the network is partitioned, leaders will be elected to deal with it by which the leader with the largest port number will be sent all the request queues (from the other leaders) and merge the other leader's requests into their queue (not including duplicates) and send this merged queue out to the backups. Under heavy load the program may slow down as it goes through all of the instructions in the queue, but it should still run correctly. Our biggest weakness with the program is the latency between the node communication since all write requests must go through the leader, the leader may not have enough time to send the data to all it's back up in time for the GET. One thing that we were proud of is that in the event that a backup accidentally forwards a request to a backup (perhaps in between pings) when that backup receives the request, they will forward that request to who they thinks is the leader and so on, not losing the message. However, our implementation should be able to handle what it needs to according to the assignment.