Joshua Lew, Larry Liu
NetIDs: jtlew2, lliu65
ECE 428 Spring 2017
April 10, 2017

MP2 - Key-Value Storage System: Design Documentation

System Design Considerations and Chord Algorithm:
In our implementation of a key-value storage system, we utilized the Chord algorithm introduced in lecture in our system. Using this algorithm, we were able to execute searches for key-value pairs in *O(logN)* time. Each node is able to dynamically join our system and then would have its own finger table entries initialized. However, during a join, finger table entries of other nodes may have to be updated as well. Thus, we modularized all of the procedures involved for when nodes join the system in order to preserve the finger table correctness. The fact that the finger tables were modularized enabled us to maintain a valid finger table at all times throughout any commands. Similarly, this was done for when a node is removed from the system (either intentionally or through a failure). However, simply updating the finger tables is not the only sets of instructions required for dynamic joining and removal of nodes. In order to preserve both correctness when searching for a key as well as approximately uniform distribution of keys across nodes, key-value pairs must be removed or copied when nodes join and fail. This was the most challenging aspect of the MP, solved by storing replicas of a key-value pair at a node's immediate successor and predecessor. Some of this logic included understanding whether a certain key-value pair is either owned by the node or is a replica that's owned by either the node's successor or predecessor. Thus, we decided to store the vm id of a key's appropriate owner in a node's local dictionary of key-value pair, enabling us to correctly transfer ownership of keys (remove and call SET on them) as well as determine which keys needed to be replicated. We did not see any advantage in alternatives, such as Cassandra, to Chord because of lack of simplicity. With Chord, maintaining an up-to date finger table is the most essential aspect for maintaining correctness. The logic around updating entries was simple; thus, we chose to execute this algorithm.

For each connection, we created two channels between nodes. Heartbeating was sent out on one channel, while other requests were made on the other channel (which we will discuss later). We initialized our socket connections in a similar manner to how we executed MP1. Thus, the networking interface in the previous MP was part of a foundation for MP2. Another considerable system design is the use of multi-threading in our program. We created separate threads for sending heartbeats and checking heartbeats. This was necessary in order to prevent other parts of our program from blocking successful failure detection. Additionally, it was important to utilize launch separate threads for processing an input message that was received from another node.

Parsing a string to find the arguments according to our message protocol is very time expensive. In the case of the batch function, a node initiating the request would receive numerous acknowledgements and forwarded requests back. Thus, multi-threading this specific process was definitely a performance enhancer.

Processing Client Commands and Protocol:
When processing messages, we utilized basic Python string parsing to examine user-typed commands as well as commands from a text file. General descriptions for how we handled each of the required functions are provided below:

`GET`:
First hashed the key name and searched for the first node containing the key. The key was repeatedly forwarded to a potential holder (done by finding next node to forward to from the finger table). If the request made its way to the designated owner of the key and it did not have it, we would return an error message to the specific request.

`SET`:
Our set request functionality worked very similar to the get command. However, if a designated node is the owner, it would first check whether the key existed in its local list. If so, it would update the value and make a `‘replicateKey’` request to both its predecessor and successor. This was done to maintain the correctness of replicas keys. If the node didn’t have it, it would create the key-value pair and replicate it at both its predecessor and successor.

`OWNER`:
Similar to how we found the designated owner of a key, when this request makes its way to the owner of a key, it would return back “Not Found” if the key did not exist or return the vm_ids of itself, its predecessor and its successor.

`LIST_LOCAL`:
This function loops through all items stored in a node’s local dictionary of keys.

`BATCH`:
This was the most difficult command for us to complete due to the potential of large bandwidth arriving towards a node that initiates the batch request. We decided to append an instruction id for each command in the first text file so that we were able to print the results to the second text file in the order they were requested. In order to prevent the commands from printing to the screen, we used a “safePrint” method to detect whether the return value for a request came from a batch.

The messaging protocol we used for passing messages is as follows:

`` op`request_vm_id`key`value`final_destination_flag`batch~ ``

`op`:
1 - set key request
2 - set key return value
3 - get key request
4 - get key return value
5 - replicate key request
6 - delete replica key request
7 - get owners request
8 - get owners return value

`request_vm-id`: Integer representing the id of the vm that initiated the request.

`key`: Unhashed key.

`value`: Value of key to be set or return value for a request.

`final_destination_flag`: Boolean representing whether we are the designated node who should execute the operation.

`batch`: Instruction id number ranging from 1 to anything.

Failure Detection:
For failure detection, we utilized heartbeating between active nodes. Heartbeats were sent on separate threads every 0.25 seconds. The heartbeat timeout was 12 seconds, where we had a separate threads listening for heartbeats and updating the last sent heartbeat in a local data structure. Multi-threading this process is important since it has the highest priority in being executed to prevent false positives. One issue debugged was the fact that if we have too many threads activated on one machine, it would throttle the performance of our threads for heartbeating; thus, would falsely declare other machines as failed.

When a failure is detected, we first removed the failed node's hashed value from all finger tables. Finger table entries were then correctly updated before rebalancing of key-value pairs. For both the one node failure and 2 simultaneous node failure cases, nodes must call set on stored replicas owned by failed nodes in order to re-replicate exactly 3 total copies of a key-value pair in the system. If failed nodes contained replicas owned by active nodes, those must be replicated as well. In our `‘handleFailure’` method, we evidently make checks on all possible scenarios and branch to the correct sets of executions to handle the failure(s).

Performance Metrics and Performance Challenges:
When considering the scalability of our system, we believe it is very scalable. It takes less than 5 seconds for a new node to join the system, in which the greater the number of correct, active nodes, a reduction in the number of keys stored at each node exists. Increasing the number of nodes also reduces the look-up time (decreasing look-up latency) for some keys as key-value pairs are distributed about equally among all nodes. Regardless of the number of nodes, our failure detection time remains the same since the heartbeating timeout remains a static number and failure detection bandwidth only increases by 2 messages per new node; thus, further promoting scalability. If we had more time to further enhance our MP functionality, we would look into optimizing failure detection more robust by using the existing error catching within TCP protocol. For example, in our MP1 we utilized this to catch and send message errors because for every message sent to another address via TCP, an acknowledgement message is sent by the receiver if it is alive. Thus, this can further reduce the time it takes for us to correctly declare a node as failed.

Resources Utilized:
Provided below is a list of major resources we referenced to throughout the course of development of MP2. Major new concepts learned that were not introduced in lecture or previous classes include: basic socket programming, python programming with more complex data structures, and TCP protocol.

General Python Documentation:
https://docs.python.org/2/howto/sockets.html

Wiki Chord Algorithm:
https://en.wikipedia.org/wiki/Chord_(peer-to-peer)

Chord Implementation:
http://www.dcs.ed.ac.uk/teaching/cs3/ipcs/chord-desc.html

In-Class Lecture Slides for Chord

Secure Hashing Python
https://docs.python.org/2/library/hashlib.html