1. Design

Overall, our key-value store server with fault detection and recovery capabilities is based on the combination of the design of our ring/ping-ack based fault detection system and ring/message forwarding key-value store server in previous MPs, with the addition of crashed machines rejoin capabilities.

We will discuss our design for the failure detection and recovery component and key-value store component separately.

Failure Detector Server

The failure detector (FD) server consists of three threads, a listener, a ping thread, and a pong thread. The listener's job is to accept join requests, either from a newly joining machine or a dead-then-revived machine. Other than the very first machine during initial setup, there will be no distinction between a newly joining and a revived machine.

During initial setup, the very first machine is going to read in a xml config file, and depending on the command line option, bind to an IP and port for its operations. Then it will traverse through all machines on the list to try to establish a connection. Being the very first machine, it will not be able to establish any connection. So it will just sit any listen to incoming connections.

Following joining machines will perform the same procedure, except it will be able to reach some machine that is alive. Upon establishing a connection between the FD servers, the joining thread sends a message denoting it declares itself with new id 0, and machine count 1, all servers in the ring receiving this message forwards it, with their own id reset at 1 higher than the one that it received, and increments the machine count by 1. This message eventually will cycle back to the newly joined machine, with a accurate machine count. It then sends this final machine count around all servers in the ring again, therefore every machine will have a updated machine count after the second round of messages.

Notice according to our algorithm, at any point during the forwarding of messages, if any new machine request a join, all machine will discard the previous message and a new round of updating id and machine count proceeds.

Upon failure of any machine, which is detected by unresponsiveness to a ping message after a certain period of time, or failure of delivery of a pong message after a certain period of time, there will be two machines detecting a failure, or two failures. This is also how we could detect two concurrent failures. Since although one machine could only actively detect one failure by not receiving a pong message, it could also passively detect a failure by not being able to deliver a pong message. Therefore, in the case of two concurrent failures, and the two failed machines are chained together, then the one in front is detected with a passive failure, and the one in the back is detected with a active failure.

The notion of active and passive failure is important when it comes to the re-establishment of the ring, with new ids and machine count.

Our algorithm orders the machine that detected a active failure to initiate a repair request to the next machine on the list in the config file. This repair request is different from a join request since the pong thread of the machine could still be working, if it has not detected a passive failure. In the repair message, the machine proposes itself with the new id 0 and machine count 1. Following machine receiving this message, if has not detected an active failure, will propose itself with new id incremented by 1 and machine count incremented by 1. If it has detected an active failure, it will break the tie with each machine's hard coded id in the config file. In the case of multiple proposed 0s, there could only be one winner, since machines' id in the config file

is unique. After a repair request cycle back to the sender with a winning status, the machine then sends out a new round of confirmation messages with the final machine count and newly updates ids.

Also notice during this repair process, if any machine fails, all machines are going to ignore their previous round of repair messages, and a new round of repair messages are sent.

Overall, the reestablishment of connections and the detection of failures among the machines are capable of being interleaved, as any such event will trigger a new set of independent messages that overwrites the previous set without complications.

Key-Value Store Server

The key-value store server has two thread, one listens for and serves the client, the other is used to replicate data to two other machines.

Upon receiving any inserted key-value pair at any server, the server hashes the key to a value between 0 and the maximum number of keys we allow to be stored, which is by default 1 million. This number could be changed in the config file easily. Then the server is going to forward the key-value pair to the appropriate server. The server responsible for serving the key-value pair stores it in memory and forward the message to the next machine with a counter. All machines receiving the message will either backup the pair and forward the message to the next machine, or store it only, depending on the counter. Since we only allow two maximum failures, there is no need for all four machines to have a backup. Therefore our counter is set at two currently, and can be changed easily upon demand.

According to our design, each machine is only responsible for one forward message backup. Instead of letting one machine forward two backup messages to two other machines, which burdens one machine and increases the chance of losing data at one machine, each the backup process is done through forwarding messages instead, which more evenly distributes the risk.

When any machine fails, the machine it connected to will duplicate all data to the next machine, and the machine connected to it also duplicates its data upon establishing a new connection. Therefore, after any failure recovery stages, all data is duplicated at least twice other than from the primary copy.

When any machine reconnects, only the machine that establishes a ping relationship to the new machine duplicates its data onto the new machine, since its data is completely identical to the one the new machine is pinging.

2. Correctness

The correctness of failure detection are inherent based on the explanation above. The correctness of our repair algorithm is ensured by our repair request, where the entire connection between machine is reset, and a new set of ids are assigned. The overlay repaired is always correct since all machine reach on consensus on both their ids and the global machine count.

The correctness of data replication is given by our active and timely replication, which is upon receiving the message. The reorganization of id responsibilities is established by the new set of ids assigned to machines in the ring. Notice after one failure, all machines will have the exact same set of data, so the id assignment to each machine does not matter as long as they are connected in a ordered ring, which they are under our algorithm.

3. Cool Application Design and Experimental Results

Our cool application is based on the movie title searching application suggested by the MP specification. To use it, after starting the first server, before starting any other server, connect a client and execute command "importmovielist file.list" will import a database containing a list of movie titles and information. Upon joining of new machines, the database is duplicated to each machine.

When a client query a word by typing "movielup word", the word is hashed to a number between 0 and maximum number of keys allowed in the system, and only one machine is responsible for that particular key. The servers in the ring are going to forward the key to the server responsible for that key, and the server is going to construct a response message with all movie titles in the database that contains the word. This response message is then relayed back to the server the client connected to, and sent back to the server.

Due to our way of load balancing, that is, all servers are aware of entire database, but servers different keywords, our load balancing performance is purely based on the quality of the hash function. We are using SHA-1 algorithm, which is good enough for our purposes.

We also conducted a latency test and we learned: based on top 10 percent of the original dataset (we use top 10 percent since the entire dataset require much longer to duplicate among the servers) our application has a base latency of 42 milliseconds in case of nonexistent or single existence keywords. Based on this base latency, we also obtained an average of 280 milliseconds latency when searching the word "the", which has 12368 appearances among the top 10 percent. And an average of 71 milliseconds latency when searching the word "a", which has 2369 appearances.

Deducting the base frequency, we have an average latency of 16.03 microseconds latency per appearance in the top 10 percent of the original dataset.

4. Team

Implementation: Dennis McWherter (dmcwhe2)

Design, testing, report: Jiageng Li (jli65)

The meta-structure of our code is documented, but due to the complexity of the FD server and the cool application, one may find our documentation not enough to fully grasp the details.