Project 3. Haystack File System

Team: Legendary
Members: You Xu (youx@andrew.cmu.edu), Lu Liu (lul3@andrew.cmu.edu)

Overview

We implemented a distributed file system for photo storage, which support upload, get, and delete operations. To reach high performance, the file system realizes only one disk operation for each upload, get or delete request.

Architecture

We basically follow the design of Haystack in the original paper (except that we don't have CDN), but select our own building blocks. We uses the following tools to build our own distributed photo file systems.

- NGINX - use reverse proxy to balance loads for web servers
- Java Undertow - used to handle HTTP requests for web servers, cache servers and storage servers
- Cassandra Cluster - used to store metadata for images and logical volumes on directory servers
- Redis Cluster - used as cache to store photos that have been recently visited

Web Server

Web servers handles all HTTP requests for uploading, getting and deleting images. For image operations, web servers firstly contact directory to get metadata about the image, like the cache url, logical volume, and physical machine dns where the image is stored. And then send HTTP request to cache server and store servers accordingly. We would describe details about each operations in section *Requests Workflow.*

To improve scalability, we use NGINX reverse proxy to handle the incoming requests to balance loads between web servers, and enable multiple web servers work at the same time.

To improve usability and scalability for cache and storage servers, we implement Restful API to enable dynamic registration and deletion for cache servers and storage servers on web servers. In this way, we don't need to restart the web server each time we want to add new storage or cache servers.

Directory

Directory servers maintains metadata for images and logical volumes. We use Cassandra cluster to store these data. Benefit from Cassandra's in-memory cache feature, we could realize only memory read and write for all our queries on directory servers. And Benefit from Cassandra's replication feature, we could easily realize replication for directory servers, which enhances our scalability and reliability.

There are two tables in directory servers. One for logical volumes, and another for image metadata. Table *Volumes* maintains all storage nodes' dns for logical volumes, and also has the writable status for each logical volume. Whenever an uploading operation happens, web server could get several writable logical volumes from directory. Table *photo_entries* maintains all metadata for images, including photo id, cache url, logical volume and storage nodes for each image.

Cache

Cache servers is responsible to send register request to directory and handle GET/DELETE photo request. If the photo is in cache, just return it. If not, cache server will send request to store server, ask for the photo data and send the result back.

To improve scalability and robustness, we use redis cluster instead of single redis server as cache. Also, redis cluster can spread the load over a greater number of endpoints which reduces access bottlenecks during peak demand.

Store

Store server is responsible to handle GET photo request sent from cache server and POST/DELETE request from web server. In the design of Store, we store photo in disks and store the index table in memory. We use Redis to store the key-value pair (index table) in memory where key of the photo id and value is the list of the offset in the volume, the size of the photo data and the delete flag.

We deploy two store servers to improve the robustness. If you want to add more replicas, just deploy the Store Server onto the machine.

Requests Workflow

Upload an image *(POST <web_server_dns>/img with image data in body)*

After receiving uploading requests, the web server would do a query on directory server to get available logical volumes, and try to do write operation these logical volumes one by one. It would send POST request to each storage nodes for uploading. If a logical volume is full, the storage servers would respond accordingly to let the web server know.

The web server would set the writable flag on Table *Volumes* as un-writable. And continue to try next logical volume. Otherwise, the storage server would store the image. When storage server receives the upload request, it will check the size of the photo to see if it the volume has enough space to store the photo. If not, it will send SERVICE_UNAVAILABLE response to the upload request. If everything goes well, the store server will save the photo and some metadata into the volume, store the index record into memory and respond 200. When the web server receive a 200 response from storage server, the uploading operation succeeds, the web server could respond 200 OK to the client, and it would respond the photo id at the same time for users to retrieve images after uploading.

Get an image *(GET <web_server_dns>/img?pid=<photo_id>)*
After receiving GET requests, the web server would ask directory for the metadata of the requested image. If no according record in directory, respond 404 to indicate that no according image stored in this image distributed file system. Otherwise, send GET request to the according cache server to get the requested image. When the cache server receive the GET request, it will check the photo id in in-memory map, if the id exists, send the corresponding value(photo data) back. If not, cache server will send a GET request to store server. The store server will check this photo in the in-memory table and if it is deleted. If the photo id exists in the table and is not deleted, store server will get photo data from the volume with the index information(offset and record size) stored in memory and send it back. If not, it will respond 404 not found. When the cache server get the response, it will check the status code. If the code is 200, cache server will store the photo data in cache with an expire time of 60s, which means if the photo is not requested in 60s, it will be deleted in memory. Then the cache server will send the result back to web server. When the web server get response from cache server, it could respond the client with the same image data.

Delete an image *(DELETE <web_server_dns>/img?pid=<photo_id>)*
After receiving DELETE requests, the web server would delete the according record on directory, and send a DELETE request to cache server. The store server will set the delete flag of this photo to false(which means the photo is deleted). The cache server will delete this photo in cache.

Challenges

Deployment is pretty hard especially when we need to deploy different tools on different machines, and each tool need to run on several different servers. Sometimes, the documents on the Internet is not clear enough that we spent a long time to figure out how to correctly configure each tool we used.

What We Learned

We experienced how to implement and deploy a distributed system, and learn that it’s not easy to build and deploy one. And in the process of building a distributed system, there are many things we need to consider, like scalability, reliability and fault tolerance, which makes it difficult. We learned the trade-offs between some tools at the beginning of the project. And we learned how to use NGINX, Cassandra and Redis, and how to build scalable distributed systems with them.

Suggestions For Lab Improvement

Andrew machines are not a clean environment for this project. When many students run their Cassandra, Redis and web servers on Andrew machines, port confliction would happen, which is frustrating when we spent a long time on figuring out which ports to use. Also, the write-up is not clear enough that it’s hard for us to know what’s the expectation for our work. I believe the lab would be better if cleaner deployment environment and clearer write-up is provided.