Recreate a needle in HeyStack: Y&Y’s picture storage

Project 3 Report

Yujie Ai (yujiea) : yujiea@andrew.cmu.edu

Yuxuan Sun (yuxuans) : yuxuans@andrew.cmu.edu

Introduction

In this project we presents HeyStack, a distributed picture store system which supports both high throughput and high scalability. HeyStack draws the idea from HayStack by dividing the system into four main components: Web Server, Directory, Cache and Store (See Figure.1). Directory is built upon Cassandra and it together with web server provides the mapping between physical volumes and logical volumes. Web server provides user interfaces and also serves as a load balancer which distributes incoming requests to different cache/store servers. Cache is built upon Redis which provides a high speed in-memory cache. Store provides persistent storage of picture files. It combines lots of relative small pictures into large haystack (i.e., volume files) and uses needles (in-memory data structure storing metadata of pictures) to reduce disk operations. In addition, Nginx is used as a reverse proxy to balance loads. Components communicate with each other via HTTP/RPC protocols and with the client via HTTP protocols .

HeyStack supports three basic file operations: upload, get, delete. When a picture file is uploaded, a unique picture id is assigned to this picture and returned to the user. The user can then use this unique identifier to get/delete this picture.

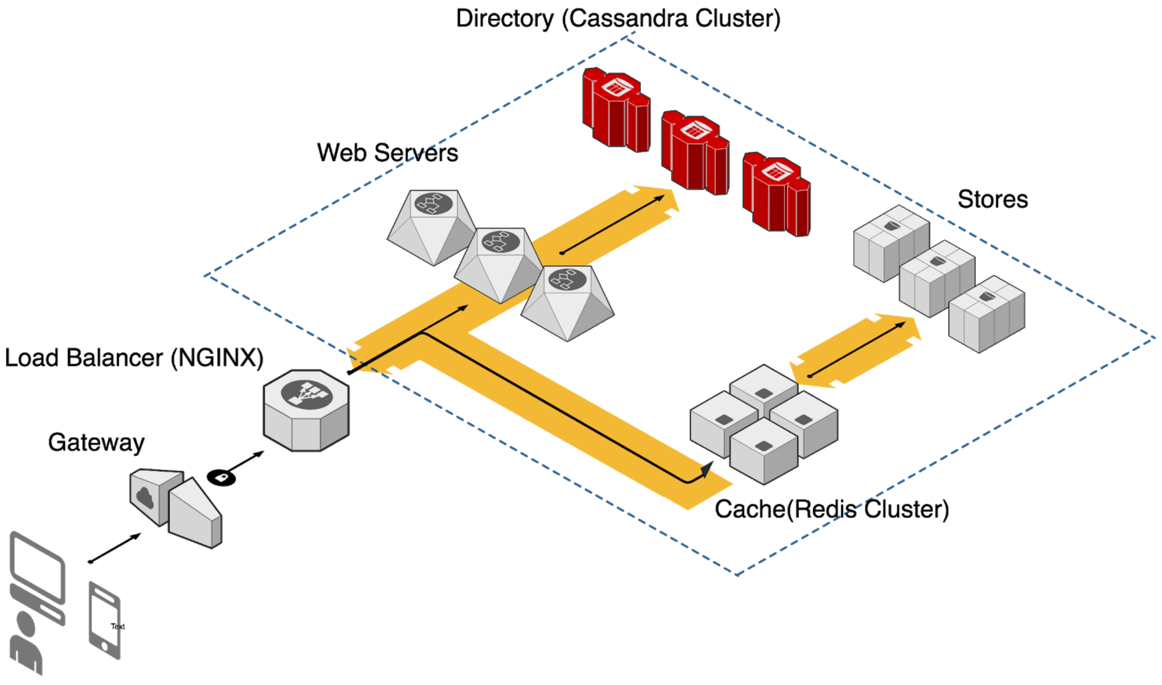

Figure.1 HeyStack Architecture

Design

1. Web server

The web server receives requests from clients, interacting with one or more components of HeyStack to handle requests, and returns results to clients. It also balances loads of incoming requests and distributes them into different nodes in the cluster.

2. Directory

HeyStack Directory is built upon Cassandra cluster and it provides a easy-to-understand and robust interface for web server to retrieve the location information of pictures. It stores the mapping from the picture id to the logical volume id, which can further be used to construct the request URL which clients will use to retrieve photos.

3. Cache

HeyStack Cache is a distributed caching system built upon Redis Cluster. It provides high-throughput, fault-tolerant caching service for this distributed photo file system and it communicates with web server, store server and the client. It maintains an in-memory hash table and uses picture ids to retrieve picture data. If a picture request is hit, it directly retrieves and returns picture data. If the request is missed, it will ask the Store Server for the picture data, add the data into its in-memory hash table and return the data to the client.

The hash table is divided by keys into several shards, each stored in a cache node. For each hash slot, we store N replicas (N can be specified) in different cache nodes. When a single node crashes, we can get data from any of the other N-1 replicas. The sharding and replication mechanisms enable a highly scalable, fault tolerant caching service which is the solid foundation of our distributed picture file system.

4. Store

HeyStack Store is an encapsulation of physical persistent storages and it serves as a high-throughput, fault-tolerant, distributed storage system. Each Store Server contains several physical volumes, each of them belongs to a logical volume. It also maintains a in-memory hash table which contains the offset, size, type and logical volume id of pictures. The physical volume is a big file which consists of many small picture data. When a store server receives a request, it first retrieves the picture id from the request, then get all the required information of this picture from the hash table. After getting all the required information, it will directly retrieve certain size of the file which is specified by the logical volume id starting from the offset. This significantly reduces the number of disk operations.

5. File Upload

Once a file is uploaded by clients, the web server accepts the HTTP request from the client and retrieves the picture data. It then generates a unique identifier (picture id) and assigns it to this picture. It also uses a hash function to determine in which logical volume should the picture reside. After generate all required information about this picture, it will communicate with the Directory and update the mappings with the newly added picture. At the same time, it will send HTTP requests to all Store servers which contains physical volumes corresponding to the picture's logical volume and ask them to store the picture. Confirmed that both Directory and Store has added this picture, it will return a success code and the picture id back to the client.

6. File Get

For get requests, the web server gets the picture id from HTTP requests and asks the Directory for the store machine id and the logical volume id. It then construct a url with the following information: Cache Server address, Store Server id and the logical volume id. It then returns the url to the client and let the client to retrieve the picture by visiting the returned url.

When the client visit the url returned by the web server. It will visit the Cache Server and ask for the picture data. The Cache Server will first check its in-memory hash table. If there's a hit, it will directly return the picture data to the client. If there's a miss, it will make request to the Store Server specified by Store Server id and retrieve the picture data from it.

7. File Delete

When the client make a Delete request, the web server gets the picture id from the request and asks the Directory for all the Store Servers where the picture is stored and the logical volume which the picture resides. It then send delete requests to all Store Servers which stores this picture and also send delete requests to Cache Server. Confirmed that all delete requests it sent are succeeded, it will return a success code to the client.

When a Store Server receives a delete request from web server, it will look into the in-memory hash table to see if there's a corresponding picture id. If there's a existing picture id, it will mark the 'is_deleted' attribute of the picture to True. It will not actually delete the picture data from the disk.

When a Cache Server receives a delete request from web server, it will look into the in-memory hash table to see if there’s a corresponding picture id. If there’s a existing picture stored in the cache, it will simply delete this picture from the cache and return success to the web server.

Consideration & Tradeoff

1. Fault tolerance

Fault tolerance is an important concern in a distributed storage system. In HeyStack, the three core components: Directory, Cache and Store, has considerable capability of fault tolerance. Directory is built upon Cassandra cluster, which provides a stable and reliable data storage as our mapping table between picture id and volume id. A single node failure won’t disable the whole Directory. Cache is built upon Redis cluster, which will periodically send heartbeat messages between nodes and detect failure. A single failed node won’t disable the Cache. The Store consists of several servers, each contains several physical volumes, each mapped to a logical volume. A logical volume is mapped to several physical volumes, which are distributed to different store servers. This serves as a replication mechanism of picture storing and a single store server crash won’t disable our capability of retrieving a picture.

2. Throughput

The needle mechanism in Store enables fast accessing to a picture, which improves the throughput of the whole system. By combining small picture data into a single large file, it reduces the metadata size so that they can be put into memory and it significantly reduces the disk operation while accessing a picture. Also, it keeps all the opened file descriptors for physical volumes and the metadata of the file in memory, which provides faster access as well.

3. Tradeoff

Considering our application scenario: written once, read often, never modified, we trade our writing performance for better read performance. Once a picture is uploaded, it is required to be written to multiple store servers, which incurs more operation. However, this provides more fault tolerance as single store server crash won't disable the whole system.

Implementation

Our HeyStack is built upon andrew machines. Four core components resides in different andrew machines and they communicate using http requests. For demonstration purpose, we use three andrew machines as our store servers, which is enough to show our physical volume & logical volume mapping mechanism and fault tolerance capability. Also, we set up three andrew machines for web server, Directory and Cache to show their functionalities.

Greatest Challenge

The greatest challenge we encountered in this project is how to concretely make a robust design of the distributed storage system. How to divide the tholw system into different functional component, how to design the interaction between these components, how to deal with the fault tolerance of each component and how to make tradeoffs between each desired capability.

Also, the setup in andrew machines also poses several challenges.

What we learned

From this project, we learned that aside from the coding challenge of real-world problem, system design is actually a different but imperative part of it. A well-designed system which provides robust functionalities, high throughput, fault tolerance and scalability is very important and thus, not easy to achieve. Before we start on any industrial challenge, we should carefully analyze the requirements, make tradeoffs based on the characteristics of our problem domain, consider the scalability and robustness of the system, and then make our design. A well-designed system will be beneficial both in short and long term.

Suggestions for next iteration of class

A quantified performance requirement may be helpful for students to evaluate their system design and guide them to make improvement.

Installation of tools and environment setup is troublesome, especially on andrew machines which we are not in the sudoer list. AWS with course specified AMI will be very helpful for students.

Reference

Beaver, D., Kumar, S., Li, H.C., Sobel, J. and Vajgel, P., 2010, October. Finding a Needle in Haystack: Facebook's Photo Storage. In *OSDI* (Vol. 10, No. 2010, pp. 1-8).