CMPE 275: Enterprise Application Development – Spring 2019

Project Report

Project: Fluffy
Professor: John Gash

SAN JOSÉ STATE UNIVERSITY

TEAM – Drop.io

Mohdi Habibi
Prathamesh Karve
Sai Harshith Reddy Gaddam
Harsh Agrawal

Table of Contents

Chapters

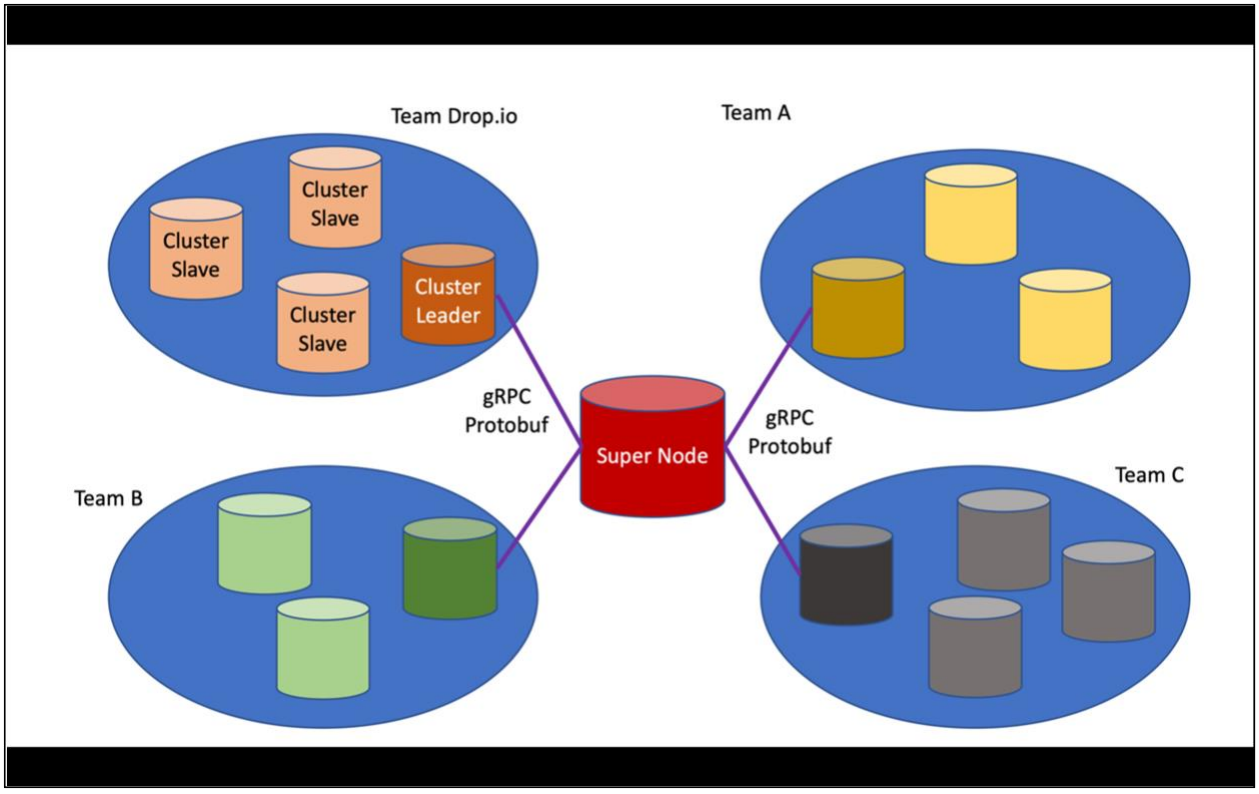
1. Project Description

The Fluffy project aims to develop a distributed data storage system. The system consists of multiple heterogeneous clusters; heterogeneous in the sense that each cluster's choice of internal design is independent of the design choices made by the other clusters. However, since the clusters together form a single service, they need to agree on the mode of communication with each other.

A user can opt to use the services of a particular cluster by talking to the cluster's end-point. Alternatively, the user can use the service through the common endpoint, called the "super node". Fluffy supports data of all forms text, structured or images.

Drop.io is the name of a cluster within Fluffy and the team that developed it. It borrows its name from an [erstwhile online file sharing service] that was bought by Facebook. We don't mind a similar fate for ourselves.

2. System Architecture



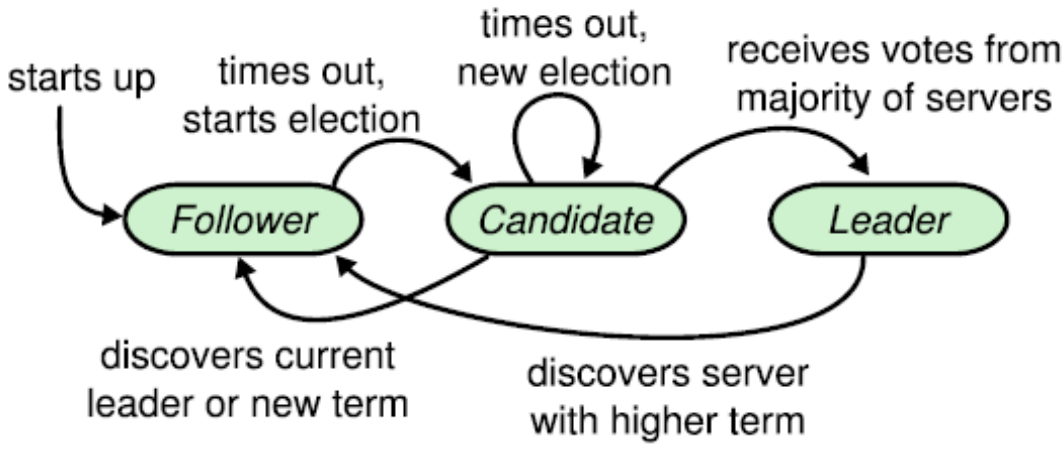
First, let's look at the internal cluster design for Team Drop.io. The cluster always has a leader node, dynamically elected among its members. The leader takes care of the control plane tasks, which are mainly cluster management and communication with the outside world. Rest of the

nodes handle the data plane. They are responsible for handling user data and providing quick access to it. All these slave nodes are symmetric in terms of the functions they carry out. So, any of these can be replaced by another similar node while the cluster is functioning. Let us look at some of the key design components in detail –

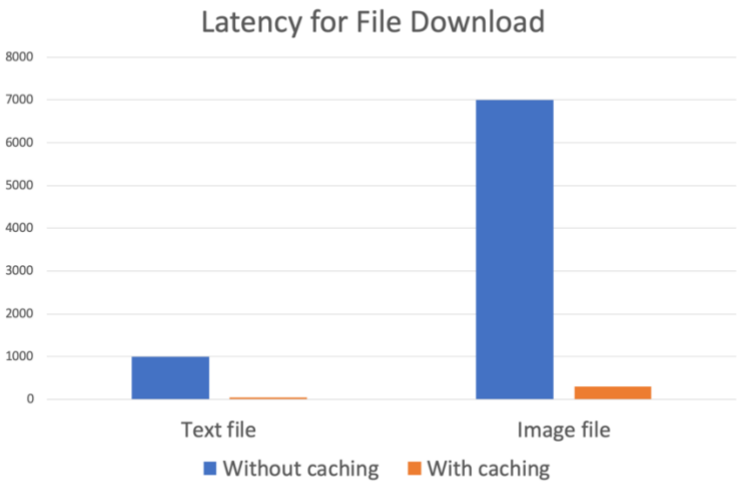
- **Cluster Leader**
  Heartbeat – The cluster leader requests the health status of each node in its cluster every second. The leader aggregates this data and notifies the cluster health to the centralized super node on behalf of the cluster using another heartbeat. This heartbeat contains the average of CPU usage, Memory, and Disk space of all the nodes in the cluster. This health data is used for work stealing.

  Data handling – When a file access request is routed to the cluster, the request is handled by the leader. If it is a request to store the file, the leader chooses the right node and routes the file to it. The leader maintains metadata about all the files that the cluster holds. The metadata helps is quick file retrieval as it points to the node which holds the file. Also, the leader can efficiently respond to file search request by the super node by just looking at its metadata, instead of checking with each member node.

- **RAFT Consensus Algorithm**
  RAFT is the algorithm we use for leader election. Initially, each node starts out with a timer of random count. Whichever node times out first, increments its term, marks itself as a candidate and starts a leader election and asks for votes from other nodes. The candidate node can vote for itself. When the candidate node receives n/2 + 1 votes (n: No of nodes in the cluster) then the candidate node becomes the leader. Otherwise, re-election occurs. In the event of leader failure, follower nodes do not receive a heartbeat from the leader to reset their timer. The node whose timer runs out first will be become a candidate node and re-election takes place.

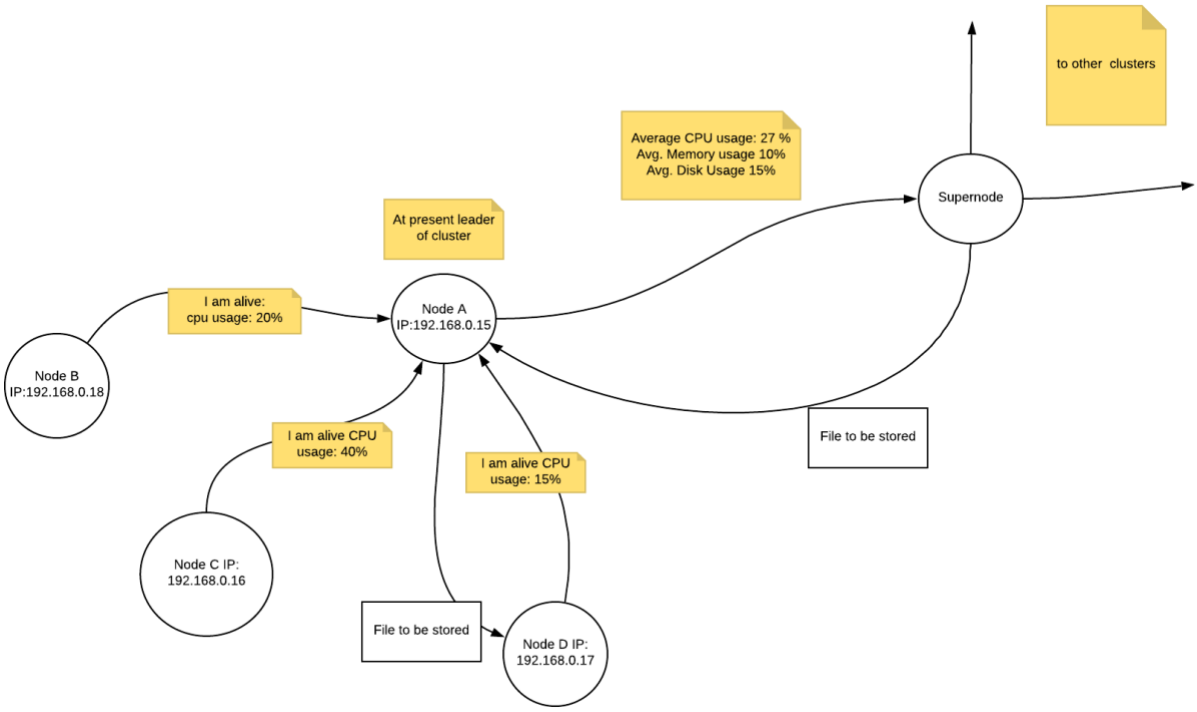


- **Caching**
  Frequent file accesses are optimized by the use of in-memory caching. All the nodes support caching, including the leader node, which uses cache for storing the metadata. Redis is used as the in-memory caching database. Any new data is entered in both Redis as well as the general purpose and more durable database MongoDB. During file search and data retrieval, each node first checks in Redis whether the data is available. If not found, it accesses the data residing in MongoDB. The use of cache improved the performance of the system in case of repeated file accesses. Below is a graph showing the latency statistics with and without caching.

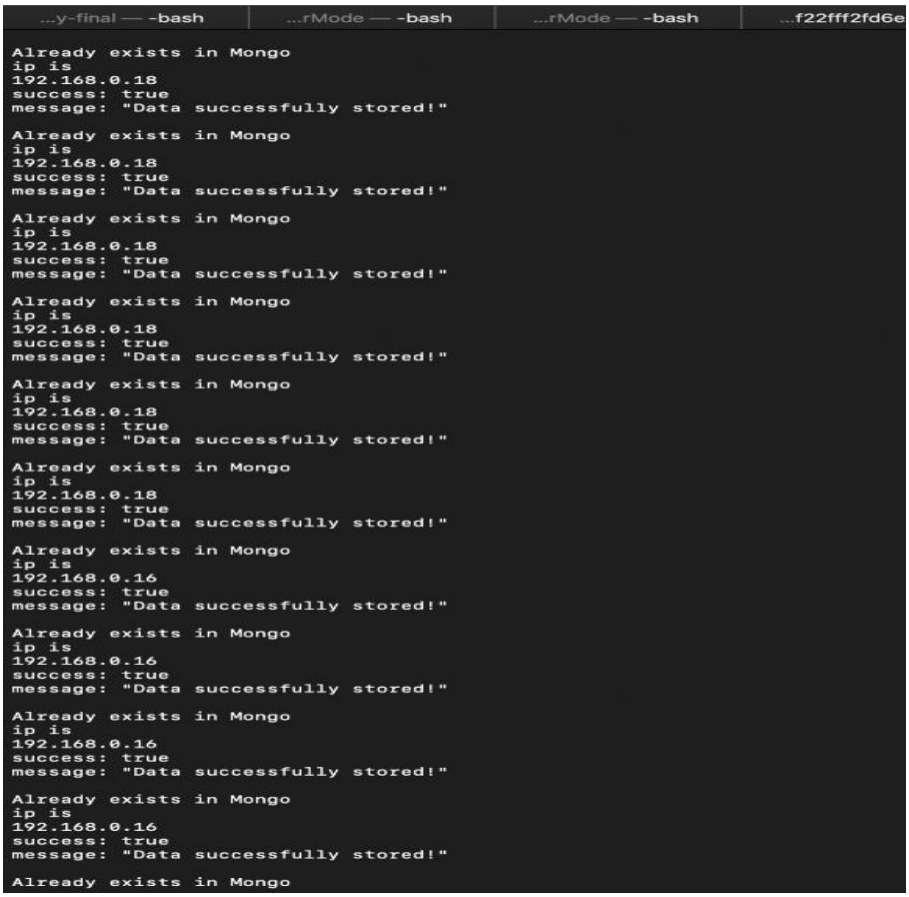


- **Work stealing**
  Work Stealing is the task scheduling strategy used within the cluster. When a nodes has no work to do or has capability to do work, it is picked up to do the task and leaving other nodes aside to complete their tasks and in this this way a node "steals" work items from other nodes.

  Cluster Status: A cluster to store data is selected is based on average of cpu usage of live nodes within cluster, average of memory usage of live nodes within cluster, average of disk usage of live nodes within cluster sent by the leader of the cluster. After the cluster is selected to store data it's on the cluster's leader to select node to store.
  Least Busy Server/node: the leader selects the node whose cpu usage is least at the moment when transfer of file occurs, instead of any random server/node to store the data.



In this way a node steals work from another by showing its CPU usage and let the leader know it can perform better and hence forth increases the performance of cluster.



As we see from the console the node to store the file is different based on the CPU usage. Node with IP 192.168.0.16 grabs the file when the node with IP 192.168.0.18 becomes busy. If the file already the particular notifies that its already stored for the user who uploaded the file and do not perform rewrite of file.

The architecture design consists of some common design choices like discovery of other clusters and the communication between them. For discovery, a centralized “super node” is used. Each cluster has a leader node which co-ordinates with the super node. Cluster to cluster communication is also handled by the leader node. Let’s look at these generic design elements in detail –

- **Super Node**
  The super node is responsible for handling client requests. The leader node of every cluster sends heartbeat to the super node which is a form of cluster discovery for the super node. The leader node sends its cluster statistics (CPU usage, Memory used, Available disk space) to the super node. When the super node receives a request (file upload) from the client it sends the request to the leader of the cluster having least CPU usage. So, this acts as work stealing at the cluster level.

- **Cluster to Cluster Communication**
  The individual clusters need to be loosely coupled to each other. But, without a common set of communication methods (the language), the clusters won't be able to work with each other. To solve this problem, we use gRPC and Protobuf. Further, the same proto file that serves in-cluster communication is used for communication between two clusters. Protobuf's extensibility allows individual clusters to add functionality to its nodes beyond what is supported by nodes of other clusters. The gRPC-Protobuf combination also provides language independence. Each cluster is free to use whatever technology they wish as long as the proto file remains constant.

3. Technologies

We used following technologies in our project:

1. Language used for the application code – Python
2. gRPC (Both inter-cluster communication and intra-cluster communication)
3. Protobuf as the interface definition language
4. Redis cache – In-memory database
5. MongoDB – Durable database

4. Performance Testing

We tested our solution with the following test cases:

1. Dynamic Leader Election when all nodes are connected
   Latency ranges from 30 seconds to 45 seconds
   Passed

2. Dynamic Leader Election when one or more nodes go down
   Latency ranges from 40 seconds to 1 minute
   Passed

3. Replication of data uploaded by client
   Passed

4. Availability of data when one or more nodes go down
   Passed

5. Uploading 5 files each of 500MB to 1GB sizes parallelly to the cluster
   Individual file upload takes about 10 seconds
   Passed

6. Downloading 100 files of smaller sizes (<50 MB) from nodes in the cluster
   Download complete in 5 seconds
   Passed

7. Getting list files from other clusters
   Passed

5. Issues faced

1. Discovery of the clusters in the network
2. Figuring out on how to assign dynamic IP to each node
3. Deciding on the common proto file to be used for the intra team communication.
4. Handling client requests simultaneously rather than in sequence.
5. Compatibility issues when tested the file transfer with other teams.
6. Running different gRPC servers at the same time.
7. Integrating and debugging the code within the team.
8. Using commands independent of the operating systems.
9. Handling re-election among the nodes using raft after network failure.
10. Using sharding to spread load on the servers
11. Replicating the data in the event of network failure.

6. Limitations and Potential enhancements

1. The super node has a static IP.
2. All the client requests go through the super node. This can create a bottleneck if the volume of requests grows larger.
3. The entire system functioning depends on the super node being up and running.
4. Cross cluster file replication and sharing can be implemented which can come in handy when one of the clusters is down.
5. Along with health statistics, we can check for clusters/nodes that are least recently used.

7. Individual Contributions

Mohdi Habibi
Implementing gRPC of file upload and download for client and server, super node heartbeat, exception handling for network failure and integrating components

Prathamesh Karve
Implementing gRPC for intern cluster communication, implementing storage using Redis cache and MongoDB database, component integration and exception handling

Sai Harshith Reddy Gaddam
Implemented raft consensus algorithm for leader election and storage of files with hashing and sharding, testing components on inter-cluster and leader responsibilities, implemented super node to select node for data transfer

Harsh Agrawal
Implemented raft consensus algorithm for leader election, functionality to split the file into chunks to store and stitching chunks for retrieval, testing components on inter-cluster and leader responsibilities, implemented super node to request and send the file