



Fluffy - Distributed Storage System

CMPE 275 Spring 2019

Instructor - Professor John Gash

Nrupa Chitley - 012483276

Prabha Veerubhotla - 013785759

Prathamesh Patki - 013772395

Vinod Katta - 012420642

Introduction

Abstract

We have implemented a distributed storage system called 'fluffy'. In the overall architecture of the system there are many clusters and each cluster communicates with the other cluster via a Super Node. Each cluster has a Master node and Slave nodes. The client interacts with the clusters via Super node. All client request like upload file, download file, update file and delete file are coordinated through the Super Node. The system is independent of the format of the incoming data and can store any type of data. The overall architecture of the system is below:

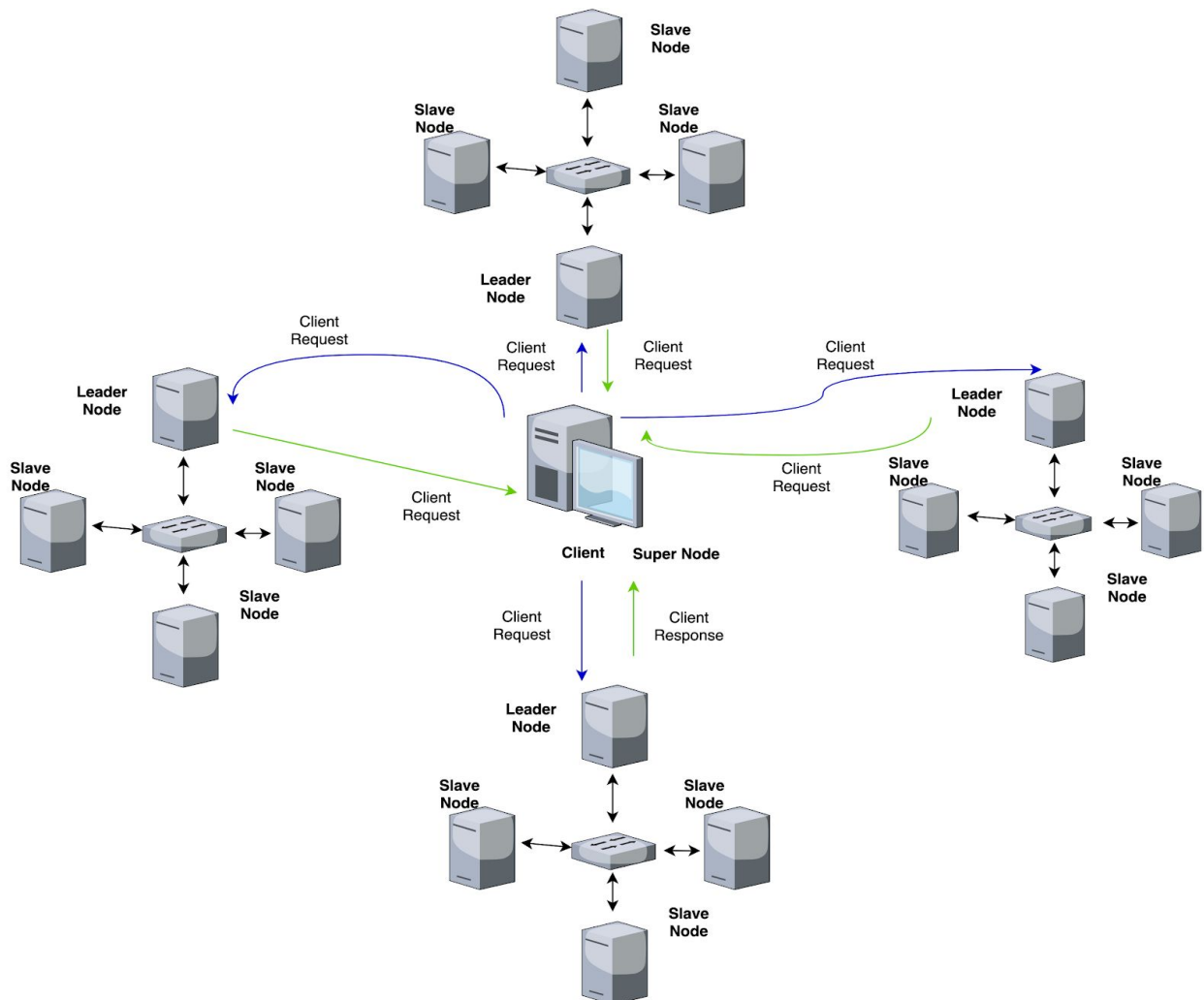


Figure 1: Full Distributed Storage System Architecture

Problem Statement

Storing data in cloud or on a network of servers share a few common challenges - how to securely share, store, and survive failures and corruption of data. Distributed data storage systems like dropbox, google drive, HDFS etc. using multiple servers and redundant solutions. There are many difficulties to store and retrieve data from a large distributed storage systems. Some of the properties of distributed storage systems are:

- Data is replicated along multiple server to avoid loss of data in case of server failure. This reduces the risk of single point of failure for the system.
- The system must be scalable to handle large traffic.
- There should single point of control like Master Node to serve requests efficiently.
- Also the there must be a backup Master Node in case of failure thus avoiding single point of failure.

Proposed Solution

In the overall scheme of the project our team represent a cluster. We have implemented Master-Slave architecture for our cluster. In our cluster the Master Node also acts as DHCP server and dynamically allocates IPs to the Slave nodes in the cluster. We have also implemented a heartbeat system to keep a track of alive and dead Slave nodes. The system is also capable of running on static IP.

The Master Node is the single point of contact for the client. Client requests are received by the Master node and then the requests are passed to the Slave nodes for completion and response. The Master Node stores the Metadata for all the files stored by the Slave Nodes. Master Node is also responsible for replicating data along 2 Slave nodes. We are using Redis to store metadata.

Slave nodes use two layers of storage, Redis and MongoDB. Data is first written to Redis when the file is being streamed and when streaming is complete the file is written to MongoDB. We are using round robin load balancing algorithm to distribute work among Slave nodes.

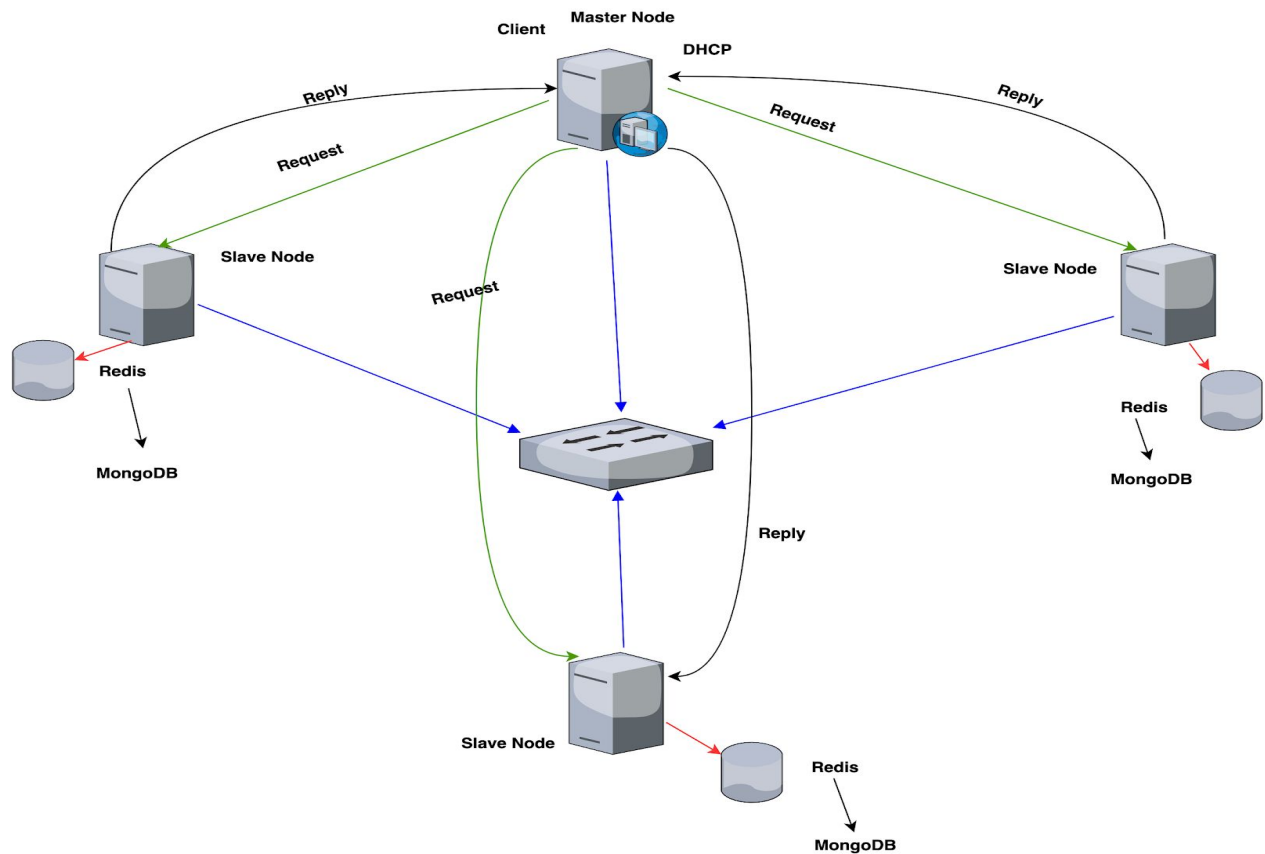


Figure 2: Cluster Architecture

Challenges And Learnings

The fluffy project was great learning experience. We faced many challenges during the process of developing the application. We have the opportunity of developing a distributed system from scratch without using any available stacks.

We exposed to many new technologies like gRPC, Protocol Buffers (protoBuf) and various distributed technology concepts like replication, RAFT leader election, Hashing etc. Apart from technical learning we learnt a great deal of non-technical skills like teamwork, collaboration, interpersonal communication and how to work in sync and collaborate with the whole class. In short we had many takeaways from this course.

While developing this system we faced many challenges, like:

1. How to stream large files gRPC and how to replicate files to another slave node simultaneously. After many brainstorming sessions we decided to have two gRPC calls (replicate and original) run simultaneously.

2. Another problem we faced was to increase the efficiency of writing to MongoDB. So to avoid multiple writes to MongoDB for same file we adopted write-through strategy with Redis cache.
3. We faced with a challenge with implementing 'updateFile' API with streaming file. So after some pondering we decided that update will be a combination of delete file and upload file.
4. We faced the difficulty of distributing the workload among team members. We also faced the challenge of integrating individual work with each other with everyone's tight schedules.

Technology Stack

Proto Buffers (Protobuf)

Protobuf is a serialization format developed by google to send data across servers. Protobuf are widely used to communicate between services. Protobuf is language and platform neutral and can be used to serialize data during communication. Protobuf provides cross language communication. Protocol buffer data is structured as *messages*, where each message is a small logical record of information containing a series of name-value pairs called *fields*. For example:

```
message Person {  
  string name = 1;  
  int32 id = 2;  
  bool has_ponycopter = 3;  
}
```

Protobuf has many advantages like, easy schema, backward compatible, cross language compatibility.

gRPC

gRPC can use protocol buffers as both its Interface Definition Language (IDL) and as its underlying message interchange format. In gRPC a client application can directly call methods on a server application on a different machine as if it was a local object, making it easier for you to create distributed applications and services.

The following are the protobuf message formats used for interaction between the nodes.

```
message FileInfo {
    string username = 1;
    string filename = 2;
}

message FileListResponse {
    string Filenames = 1;
}

// The request message containing ip address and leader status
message ClusterInfo {
    string ip = 1;
    string port = 2;
    string clusterName = 3;
}

// The response message containing the status of the server
message ClusterStats {
    string cpu_usage = 1;
    string disk_space = 2;
    string used_mem = 3;
}

message Empty {}

message NodeInfo {
    string ip = 1;
    string port = 2;
}

message NodeName {
    string name = 1;
}

message UpdateMessage {
    string message = 1;
}

message DataType {
    string username = 1;
    string filename = 2;
    string type = 3;
}
```

```
}
```

Redis

Redis is an open source, in-memory data store. It supports many data structures few of which are strings, hashes, lists, sets, sorted sets, bitmaps. Redis can hold complex data structure as value for a specified key. In this project, we are using Redis at two places - to store metadata in Master node and as cache layer in Slave nodes. We are using JedisPool, a thread safe pool of Redis connectors to serve requests at a faster rate.

MongoDB

MongoDB is a NoSQL database and observes BASE (Basically Available, Soft State, Eventually Consistent) properties. It uses JSON like documents to store data. We have used MongoDB as our main storage system as it supports flexible schema and is highly scalable and available.

System Design

DHCP

We are using DHCP server to allocate IPs dynamically to all Slave nodes. This helps in having different IPs for different nodes and also, we were able to track of newly added nodes in the network and the system can be in sync. DHCP server is run on the Master node. A thread is running continuously to monitor the DHCP lease file for any changes.

Code for monitoring lease file:

```
public void monitorLease(boolean dhcp) {
    isDhcp = dhcp;
    if(!isDhcp) {
        return;
    }
    //Check for changes in dhcpd lease file
    TimerTask task = new Dhcp_Lease_Changes_Monitor(new
    File("/var/lib/dhcpd/dhcpd.leases")) {

        protected void onChange(File file) {
```

```

        // here we code the action on a change
        try {
            //TODO: replace the command with relative path or use root
            dir

            logger.info("Calling onchange dhcpd.lease file");
            Process p = new
ProcessBuilder("/home/vinod/cmpe275/demo1/275-project1-demo1/fetch_ip.sh").
start();

            BufferedReader reader1 = new BufferedReader(new
InputStreamReader(p.getInputStream()));
            newIpList.clear();
            logger.info("new ip list: " + newIpList.toString());
            String output;
            while ((output = reader1.readLine()) != null) {
                newIpList.add(output);
            }

            compareAndUpdate();
            copyList();

        } catch (IOException | StatusRuntimeException io) {
            logger.info("Exception while handling changes of lease file:
" + io);
        }

    }
};
Timer timer = new Timer();
// repeat the check every second
timer.schedule(task, new Date(), 1000);
}

```

Client API

Java API, Python API

We have implemented the client API in both Java and Python. As, the gRPC protobuf works across languages and platforms, the client and server can be any language, independent of one another. The gRPC automatically generate idiomatic client and server stubs for your service in a variety of languages and platforms. We support the following operations from client side:

- Put - save a file
- Get - retrieve a file
- Update - update the existing file(already saved in server)
- Search - search if the given file exists
- Delete - delete the file
- List - list all the files of a given user

Leader Election

We have implemented very easy leader election algorithm. When the server is spun up, all the nodes will send their IP information to each other using 'sendVote()' method. Once the node has received all the votes. The voting will happen on grounds of which node has the highest IP and that IP will be declared as Leader.

```
public NodeInfo electLeader(NodeInfo node1, NodeInfo node2){
    if(node1 == null)
        return node2;
    String[] node1List = node1.getIp().split(".");
    String[] node2List = node2.getIp().split(".");
    if(Integer.parseInt(node1List[node1List.length - 1]) <
Integer.parseInt(node2List[node2List.length - 1])){
        return node1;
    }
    return node2;
}

@Override
public void vote(NodeInfo request, StreamObserver<ack> responseObserver) {
    super.vote(request, responseObserver);
    ips.add(request);
    if (ips.size() == NODES) {
        for(NodeInfo ip: ips){
            leaderNode = electLeader(leaderNode, ip);
        }
    }
}

public void sendVote() {
    logger.info("getting current ip list");
    List<String> currentIpList = new ArrayList<>();
```

```

try {
    Scanner s = new Scanner(new File("../../../raftIPList.txt"));
    while (s.hasNextLine()){
        currentIpList.add(s.nextLine());
    }
    s.close();
} catch (FileNotFoundException e) {
    e.printStackTrace();
}

for(String ip: currentIpList) {
    Node_ip_channel node_ip_channel = new Node_ip_channel();
    node_ip_channel.setIpAddress(ip);
    ManagedChannel ch = ManagedChannelBuilder.forAddress(ip,
Integer.parseInt(nodePort.trim())).usePlaintext(true).build();
    node_ip_channel.setChannel(ch);
    nodeIpChannelMap.put(ip, ch);
}

nodeIpChannelMap.forEach((ip, channel1)->{
    blockingStub=FileserviceGrpc.newBlockingStub(channel1);
    NodeInfo.Builder nodeInfo=NodeInfo.newBuilder();
    nodeInfo.setIp(ip);
    nodeInfo.setPort(nodePort);
    blockingStub.vote(nodeInfo.build());
});
}

```

Master Node Implementation

HeartBeats

We are using heart beats to get the health statistics for all slave nodes. The thread implements this task every 60 seconds and updates the list of currently available slave node IPs. This helps in keeping a track of dead and alive slave nodes.

```

private void getSlavesHeartBeat() {
    logger.info("Started monitoring heart beat of slaves..");
    TimerTask timerTask = new TimerTask() {
        @Override
        public void run() {
            logger.info("running heart beat monitoring service...");
            if (dhcp_lease_test.getCurrentIpList().size() > 0) {

```

```

        try {
            Map<String, ClusterStats> clusterStatsMap =
MasterNode.getHeartBeatofAllSlaves();
            for(Map.Entry<String, ClusterStats> m : clusterStatsMap.entrySet()) {
                logger.info("ip: "+m.getKey());
                logger.info("cpu: "+m.getValue().getCpuUsage()+ " mem:
"+m.getValue().getUsedMem()+" disk: "+m.getValue().getDiskSpace());
            }
        } catch (StatusRuntimeException sre) {
            logger.info("Exception: " + sre + " while trying to get heart beat from
slaves");
        }
    }
}
};
Timer timer = new Timer();
timer.scheduleAtFixedRate(timerTask, 0, 60000);
}

```

We are also sending cluster statistics to the Super node every 60 seconds.

```

@Override
public void getClusterStats(Empty empty, StreamObserver<ClusterStats>
statsStreamObserver) {
    logger.info("got request for cluster stats");
    Map<String, ClusterStats> nodeStats =
MasterNode.getHeartBeatofAllSlaves();
    double averagemem = 0.0;
    double averagecpu = 0.0;
    double averagedisk = 0.0;
    for (Map.Entry<String, ClusterStats> m : nodeStats.entrySet()) {
        averagecpu = averagecpu +
Double.parseDouble(m.getValue().getCpuUsage());
        averagemem = averagemem +
Double.parseDouble(m.getValue().getUsedMem());
        averagedisk = averagedisk +
Double.parseDouble(m.getValue().getDiskSpace());
    }
    averagecpu = averagecpu / nodeStats.size();
    averagemem = averagemem / nodeStats.size();
    averagedisk = averagedisk / nodeStats.size();

    logger.info("average cpu usage of cluster: " + averagecpu);
    logger.info("average mem usage of cluster: " + averagemem);
    logger.info("average disk usage of cluster: " + averagedisk);
}

```

```

ClusterStats.Builder clusterStats = ClusterStats.newBuilder();
clusterStats.setCpuUsage(String.valueOf(averagecpu));
clusterStats.setDiskSpace(String.valueOf(averagedisk));
clusterStats.setUsedMem(String.valueOf(averagemem));

logger.info("Sending cluster stats...");
statsStreamObserver.onNext(clusterStats.build());
statsStreamObserver.onCompleted();
}

```

Metadata

Metadata is stored in Master node. We are using Redis to store metadata. Metadata stores a map for user, files and slave node IPs. This information is updated when a file is uploaded, file is updated and file is removed. Also the before the 'download' request for file is processed the metadata is searched to the IPs of the slave nodes where the file is saved. The metadata also served requests of listing all the files by a user. It also gives a check as to if a file exists in the storage system or no.

The data structure for metadata:

```
{userName : { fileName : [IP1, IP2] }}
```

Search file if exists:

```

/**
 * Check if file exists before PUT, UPDATE, REMOVE, GET, SEARCH
 * @return
 */
public boolean checkIfFileExists(String userName, String fileName){
    byte[] userNameByte = serialize(userName);
    try{
        if(redisConnector.exists(userNameByte)){
            byte[] val = redisConnector.get(userNameByte);
            Map<String, List<String>> userFilesMap = (Map<String,
List<String>>)deserialize(val);
            if(userFilesMap.containsKey(fileName)){
                return true;
            }
        }
    }
    else {

```

```

        logger.info("User does not exists!");
    }
} catch (Exception ex){
    ex.printStackTrace();
}
return false;
}

```

List all files for a person:

```

/**
 * Get all files for a user from MetaData
 *
 * @param userName
 * @return
 */
@SuppressWarnings("unchecked")
public Set<String> getAllFiles(String userName) {
    byte[] userNameByte = serialize(userName);
    try {
        if (redisConnector.exists(userNameByte)) {
            byte[] val = redisConnector.get(userNameByte);
            Map<String, List<String>> allFiles = (Map<String, List<String>>)
deserialize(val);
            logger.info("All user Files: " + allFiles);
            return allFiles.keySet();
        } else {
            logger.info("User not present");
            return null;
        }
    }
    catch (Exception ex){
        ex.printStackTrace();
    }
    return null;
}

```

Round Robin

We have implement round robin scheduling algorithm to distribute requests among all Slave Nodes. 'roundRobinIP' will be called before any request is processed by the Master Node. This method will return a Slave node IP which will be used to process the request. Before the 'roundRobinIP' method is called the

health of all Slave nodes is checked and only the alive Slave nodes are considered for round robin.

```
public synchronized static String roundRobinIP() {
    logger.info("current ip list: " + new
Dhcp_Lease_Test().getCurrentIpList());
    List<String> currentList = new Dhcp_Lease_Test().getCurrentIpList();
    NOOFSHARDS = currentList.size();
    logger.info("number of shards: " + NOOFSHARDS);
    currentIP = currentList.get(currentIPIxd);
    currentIPIxd = (currentIPIxd + 1) % NOOFSHARDS;
    logger.info("Returning ip: " + currentIP);
    return currentIP;
}
```

Replication

In our cluster we are replicating data along 2 Slave Nodes. When a new file is uploaded or a file is updated the two request calls - replication call and the original call, run simultaneously. This gives higher efficiency. Before replication, 'roundRobinIP' method is called twice to get the 2 Slave node IPs.

Code snippet for replication:

```
public synchronized static String roundRobinIP() {
    logger.info("current ip list: " + new
Dhcp_Lease_Test().getCurrentIpList());
    List<String> currentList = new Dhcp_Lease_Test().getCurrentIpList();
    NOOFSHARDS = currentList.size();
    logger.info("number of shards: " + NOOFSHARDS);
    currentIP = currentList.get(currentIPIxd);
    currentIPIxd = (currentIPIxd + 1) % NOOFSHARDS;
    logger.info("Returning ip: " + currentIP);
    return currentIP;
}
```

Data Storage

Redis DB

Redis is used in every Slave node as cache. We are using write-through cache strategy to reduce the cost of multiple writes to MongoDB. In write-through strategy cache sits inline with the database and write is first written to cache and

then to database. We are streaming the file coming from the client, thus for each file upload there are multiple calls made to the Slave node by the Master node. Each chunk of data streamed by the Master node to Slave node is first written the Redis DB. When the file is done streaming, a 'OnComplete' function is called then this file is written to MongoDB. Also Redis DB supports 4 operations: file upload, file download, file delete and file update

Code to upload file to Redis DB:

```
public String put(@NotNull String userName, @NotNull String fileName, String seqID, byte[]
content) {
    logger.info("Inside PUT redis handler");
    byte[] userNameByte = serialize(userName);
    try {
        if (redisConnector.exists(userNameByte)) {
            byte[] val = redisConnector.get(userNameByte);
            Map<String, Map<String, byte[]>> map = (Map<String, Map<String, byte[]>>)
deserialize(val);
            if(map.containsKey(fileName)){
                Map<String, byte[]> map1 = map.get(fileName);
                map1.put(seqID, content);
            }
            else{
                Map<String, byte[]> map2 = new HashMap<>();
                map2.put(seqID, content);
                map.put(fileName, map2);
            }

            String res = redisConnector.set(userNameByte, serialize(map));
            if (res == null) {
                logger.info("Error storing in Redis " + userName);
                return null;
            }

        } else {
            HashMap<String, Map<String, byte[]>> newMap = new HashMap<>();
            Map<String, byte[]> innerMap = new HashMap<>();
            innerMap.put(seqID, content);
            newMap.put(fileName, innerMap);
            String res = redisConnector.set(userNameByte, serialize(newMap));
            if (res == null) {
                logger.info("Error storing in Redis for first time " + userName);
                return null;
            }
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return fileName;
}
```

```
}
```

Code to upload file to Mongo DB:

```
@Override
public String put(String userEmail, FileEntity file) {
    logger.info("Inside PUT mongo handler");
    logger.info("userEmail: " + userEmail);
    try {
        BasicDBObject findQuery = new BasicDBObject("personEmail", userEmail);
        FindIterable<Document> temp = collection.find(findQuery);
        logger.info("Checking if data exists");
        if(temp.first() != null){
            logger.info("Inside if ");
            BasicDBObject listItem = new BasicDBObject("allData", new
BasicDBObject("fileName", file.toString()).append("value",file.getFileContents()));
            BasicDBObject updateQuery = new BasicDBObject("$push", listItem);
            collection.updateOne(findQuery, updateQuery);
        }
        else {
            List<BasicDBObject> allData = new ArrayList<>();
            allData.add(new BasicDBObject("fileName", file.toString()).append("value",
file.getFileContents()));
            Document doc = new Document("personEmail", userEmail)
                .append("allData", allData);
            collection.insertOne(doc);
            logger.info("Success " + userEmail);
        }
    } catch (Exception ex) {
    }
    return file.getFileName();
}
```

MongoDB

We are using MongoDB as our main data storage system. All files belonging to a particular user are stored in one document in the DB. We have collection known as 'Files' in 'fluffy' database. MongoDB also supports 4 operations: file upload, file download, file delete and file update. All the file contents are stored as bytes in the database.

The data structure for metadata:

```
{'personEmail': 'xyz', 'allData': [ { 'fileName': 'test.txt', 'value': { seqID, content} } ] }
```


MongoDB code for download file:

```
@Override
public FileEntity get(@NotNull String email, @NotNull String fileName){
    logger.info("Inside GET mongo handler");
    try {
        BasicDBObject elementQuery = new BasicDBObject("fileName", fileName);
        BasicDBObject query = new BasicDBObject("allData", new BasicDBObject("$elemMatch",
elementQuery));
        query.put("personEmail", email);
        Document doc = collection.find(query).first();
        logger.info("Query Successful");
        List<Document> dataList = (List<Document>)doc.get("allData");
        if(dataList.size() == 0)
            return null;
        for( Document docu : dataList){
            String checkFile = (String) docu.get("fileName");
            if(checkFile.equals(fileName)){
                logger.info("Got file: " + checkFile);
                return new FileEntity(checkFile, docu.get("value"));
            }
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return null;
}
```

MongoDB code for remove file:

```
@Override
public void remove(@NotNull String email, @NotNull String fileName){
    logger.info("Inside REMOVE mongo handler");
    BasicDBObject query = new BasicDBObject("personEmail", email);
    BasicDBObject update = new BasicDBObject("allData", new
BasicDBObject("fileName", fileName));
    collection.updateOne(query, new BasicDBObject("$pull", update));
}
```

MongoDB code for update file:

```
@Override
public boolean update(@NotNull String email, @NotNull FileEntity file){
    logger.info("Inside UPDATE mongo handler");
    BasicDBObject query = new BasicDBObject("personEmail",
email).append("allData.fileName", file.getFileName());
```

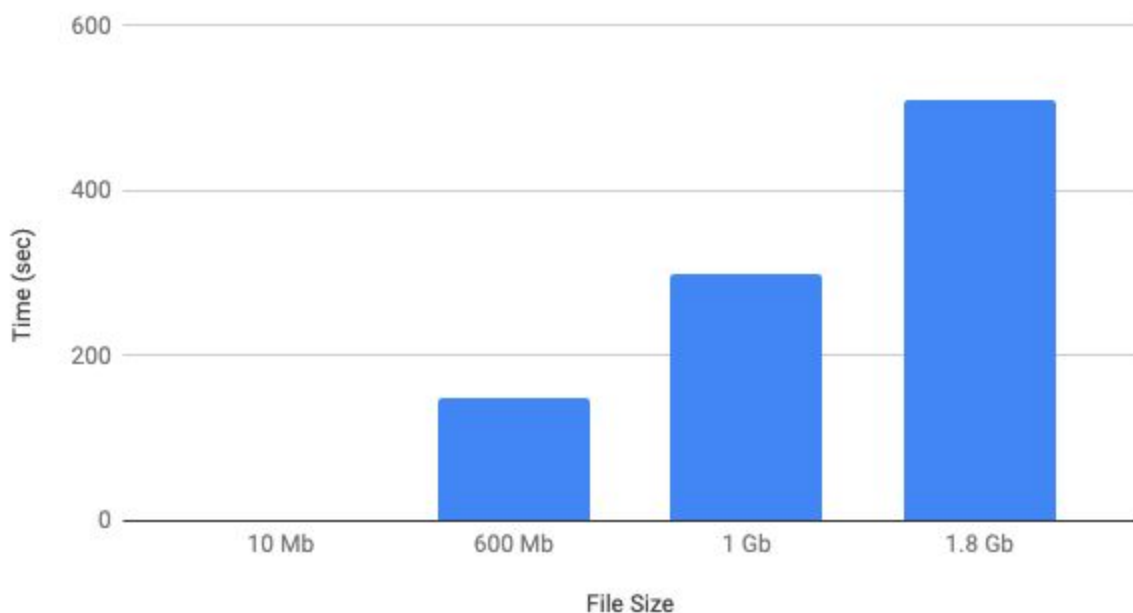
```
BasicDBObject update = new BasicDBObject();
update.put("allData.$.value", file.getFileContents());
collection.updateOne(query, new BasicDBObject("$set", update));
return true;
}
```

Milestones

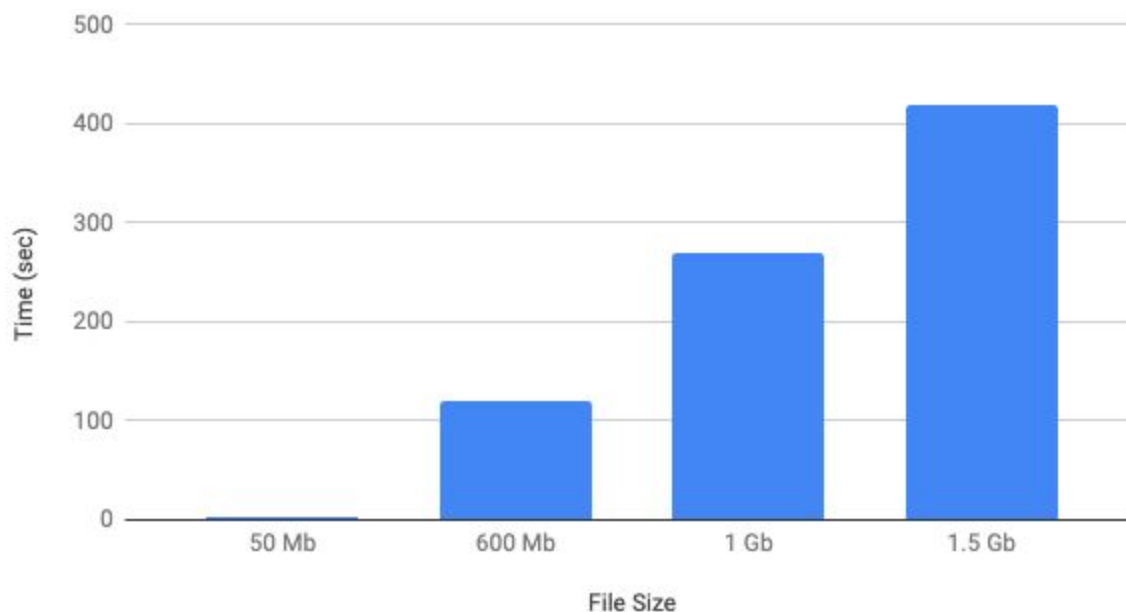
- CRUD operations for files through Java client
- In-memory DB(Redis) and backup DB(Mongo)
- Heartbeat to slaves
- Dead Node detection
- RAFT leader election
- Data replication(currently we support 2 replicas)
- Optimizing search, list operations using master's metadata
- Sharding using round robin
- Integration with super node
- Python client file feeding API

Time Analysis:

File Upload Times



File Download Times




Individual Contributions

Prabha Veerubhotla

Implemented all the rpc services for all file operations PUT(for saving the file in the server), GET (retrieving the file from the server), DELETE(deleting an existing file from server), SEARCH(searching if the file exists in the server), LIST(listing the files of a given user), UPDATE(updating the already saved file), REPLICATE(replicating the file onto the slaves) . This includes the server side implementation of each rpc call and the client side java stub calls. Added the code in the master node to check, if a node is alive, before assigning any request to it. Integration with supernode: sending leader information and the cluster stats to the super node every 6s. Added python client feeding API, to support client in both python and Java. Added client tests for all CRUD operations.

Vinod Katta

Set up DHCP on ubuntu, and configured it to assign ips in a given range. Added the data migration capability to the system, when a node goes down(not integrated completely). This helps to retrieve data from a node that goes offline. Included the



capability of detecting deadnode through heartbeat. When a node goes offline then it is no more considered for CRUD operations. Included functionality to store metadata with respect to IP address so that the files present in a dead node can be easily known and duplicated with other available nodes.

Nrupa Chitley

Developed PUT, GET, REMOVE, UPDATE APIs for Redis and MongoDB which are used for uploading file, downloading file, deleting file and updating file respectively. Developed Slave node handlers in SlaveNode.java which receive requests from Master node and process them appropriately based on file availability in Redis cache. Also developed APIs for Master Metadata to handle requests like list all files for a user (getAllFiles()) and check if a file exists or not (checkIfFileExists()). APIs like 'putMetaData', 'getMetaData', 'deleteFileFromMetaData' are used to add, get and delete metadata. Developed Round Robin load balancing algorithm code and Leader Election code. Wrote unit tests for Redis and MongoDB APIs.

Prathamesh Patki

Developed the base code for the heartbeat sending system stats from the slave nodes to the master node. This helps the master node determine the slave node to which to send the files uploaded by the client. Contributed to testing the application as a cluster, while implementing DHCP, monitoring the data transfers done by the master and super nodes, and general support activities.