FLUFFY- DISTRIBUTED FILE STORAGE SYSTEM

Submitted By:

Balamurugan Avudaiappan(010818717)

Mukul Bajpai(010810449)

Saikrishnan Baskaran(010720489)

Submitted To:

Professor John Gash

Computer Engineering Department

San Jose State University

Table of Contents

Introduction:

The objective of this project is to facilitate a fault-tolerant, scalable and reliable “Distributed File Storage System” using asynchronous communication frameworks.

Reliability is facilitated by having master-slave like architecture where there is a leader node in the cluster that never goes down (unless all nodes are down).

Fault-tolerance is facilitated by electing a new leader node in the distributed system whenever the current leader goes down.

Scalability is facilitated by using a leader as the first point of contact to do the work and then send the appropriate requests to other clusters.

Work-Stealing by the followers and **Work-Sharing** which is initiated by the leader is implemented for load-balancing (Leader in this case because most of the work is done by the leader in our architecture).

Technologies Used:

Uploading and retrieving the image is the one of the main use cases for the system and hence we should exchange data in format that is light, efficient and easy to use is required. Google's Protobuf was a good fit for it, as it provides the flexibility to define the structure of the data, we can read and write our structured data to and from variety of languages.

Handling the number of asynchronous requests is another main use case of the system. We used Netty server for designing a system that is distributed. Netty is an NIO Client server framework which enables quick and easy development of network application such as protocol server's and clients. It greatly simplifies and streamlines network programming such as TCP and UDP socket server development.

1. Google Protobuf

Protocol buffers are language and platform indifferent and provide the practicable way of serializing the structured data which is used in the communications protocol and storage. They are flexible, potent in ensuing structured data.

Problems Solved by Protocol Buffer:

Protocol buffers were developed to solve many of the problems. We can easily initiate the newly created fields, and there are certain transitional servers which do not need to inspect the data could plainly parse it and pass through the data non-essentially knowing about all the fields. The formats that we have are more implicit and we can deliver them with different formats.

2. Netty

Netty is a java library and API or NIO client server framework used to built highly concurrent networked application. It's different from standard java API as it provides asynchronous functionality to your application which means there is no return value and invocation of each method is instantaneous. Another thread will handle the result and will deliver it back to the calling method. Let's understand this by taking an example. Assume that, a client API is requesting a server to acquire the number of items from the server. So here, we can see that this method does not have any value to return. But it accepts a response handler as an argument and then listener will be called back when the item count has been acquired and then listener can do the required operations with the result. This is useful concept because it saves the resource of client as thread does not need to wait till it receives the response from server.

Design Approach

1. Intra-cluster Model:

We have used two **DNS Server** for the client interaction with the other nodes in the cluster. It helps client to depict the hostname and port of the current Leader of the cluster. The diagram given below gives the overview of the client interaction with node design model.

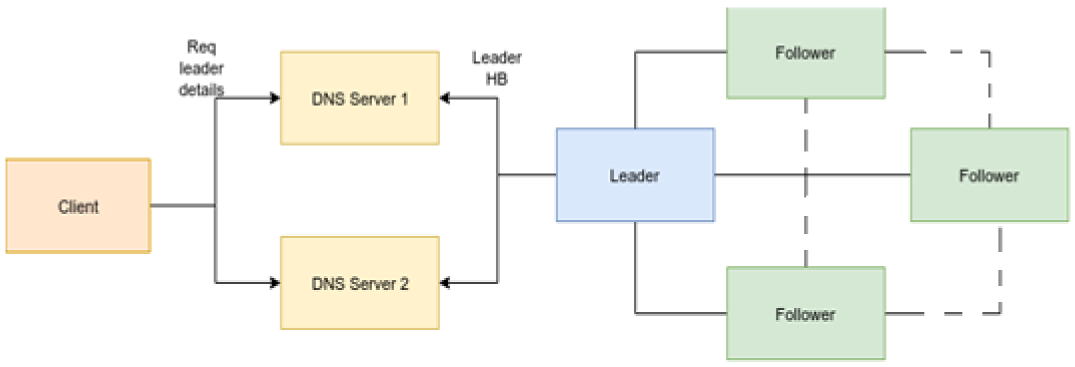


Figure 1: Client-Server Model

Assumption: If all the DNS servers go down the entire system is considered to be unavailable

Working:

1. Initially, when the server starts, one of the node initiate leader-election. Once the leader is declared, it sends the leader details to the DNS servers.
2. The DNS servers will receive the leader details after a particular timestamp from current leader, to keep it up-to-date.
3. The client sends a request to the one of the DNS servers. It this DNS server active, it will send response to client with leader details. In-case, it is found failed/not-working, then the client will contact to other DNS server and likewise it will get the leader details in response (as shown in figure 1)
4. After the leader information is available to the client, the client sends its request to the leader. Request can be read/write/delete/update.
5. The requests are sent from client to leader in command port using CommandMessage (in pipe.proto) and for leader to follower or follower to follower communication is done using the work port(in work.proto) as shown in figure 2.

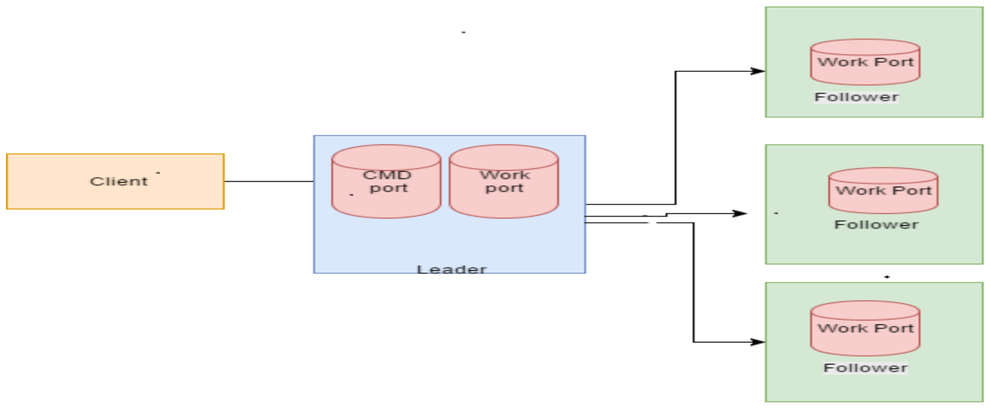


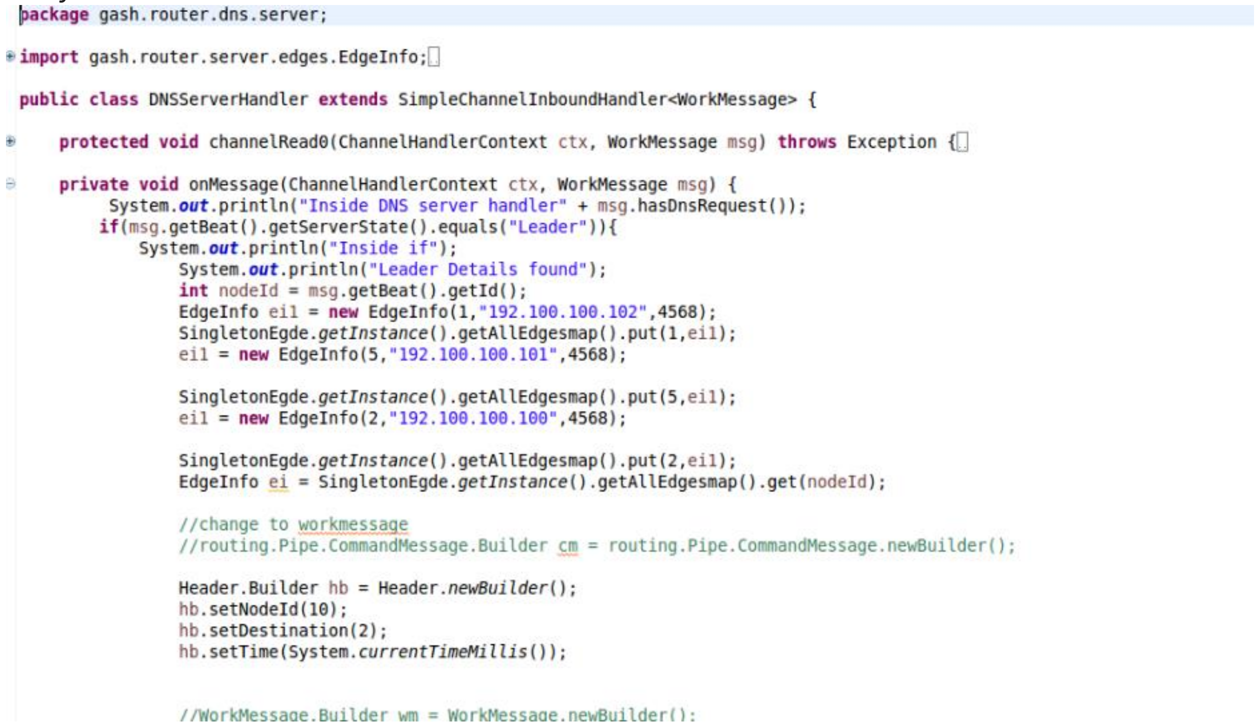
Figure 2: Intra-cluster communication

<u>Defence of Design:</u>

1. We used 2 DNS Servers to avoid single point of failure. If one DNS server fails, other DNS server will serve the request and will have the details of current leader

2. Alternate design without DNS server – Client can broadcast its request to all the nodes in cluster. The current leader will send back the response or only the leader will handle the request in this case.

We have not used the above design and choose to design DNS server to improve the performance and to avoid the network traffic-load. In-case of DNS server, we are not only avoiding the un-necessary overload to the server, but also making the application work faster and fault tolerant.



```
package gash.router.dns.server;

import gash.router.server.edges.EdgeInfo;

public class DNSServerHandler extends SimpleChannelInboundHandler<WorkMessage> {

    protected void channelRead0(ChannelHandlerContext ctx, WorkMessage msg) throws Exception {

    private void onMessage(ChannelHandlerContext ctx, WorkMessage msg) {
          System.out.println("Inside DNS server handler" + msg.hasDnsRequest());
        if(msg.getBeat().getServerState().equals("Leader")){
            System.out.println("Inside if");
                System.out.println("Leader Details found");
                int nodeId = msg.getBeat().getId();
                EdgeInfo ei1 = new EdgeInfo(1,"192.100.100.102",4568);
                SingletonEgde.getInstance().getAllEdgesmap().put(1,ei1);
                ei1 = new EdgeInfo(5,"192.100.100.101",4568);

                SingletonEgde.getInstance().getAllEdgesmap().put(5,ei1);
                ei1 = new EdgeInfo(2,"192.100.100.100",4568);

                SingletonEgde.getInstance().getAllEdgesmap().put(2,ei1);
                EdgeInfo ei = SingletonEgde.getInstance().getAllEdgesmap().get(nodeId);

                //change to workmessage
                //routing.Pipe.CommandMessage.Builder cm = routing.Pipe.CommandMessage.newBuilder();

                Header.Builder hb = Header.newBuilder();
                hb.setNodeId(10);
                hb.setDestination(2);
                hb.setTime(System.currentTimeMillis());


                //WorkMessage.Builder wm = WorkMessage.newBuilder();
```

Snapshot: DNS Server DNSServerHJandler.java.

Command Proto file format is :

```
message CommandMessage {
   required GlobalHeader globalHeader = 1;
  oneof payload {
  bool ping = 2;
  string message = 3;
  Failure failure = 4;
  Request request = 5;
  Response response = 6;
  DNSRequest dnsrequest =7;
  }
}
message DNSRequest{
required int32 host = 1;
required int32 port = 2;
}

message File {
  optional int32 chunkId = 1;
  optional bytes data = 2;
  required string filename = 3;
  optional int32 chunkCount = 5;
}

enum RequestType {
  READ = 1;
  WRITE = 2;
  UPDATE = 3;
  DELETE = 4;
}
```

Data flow in each Operations :

1.1 File Write Operation : The file write operation from the client side to the servers is occurring in form of file chunks each of size 1 MB. This decreases the load on the network.

Design Approach:

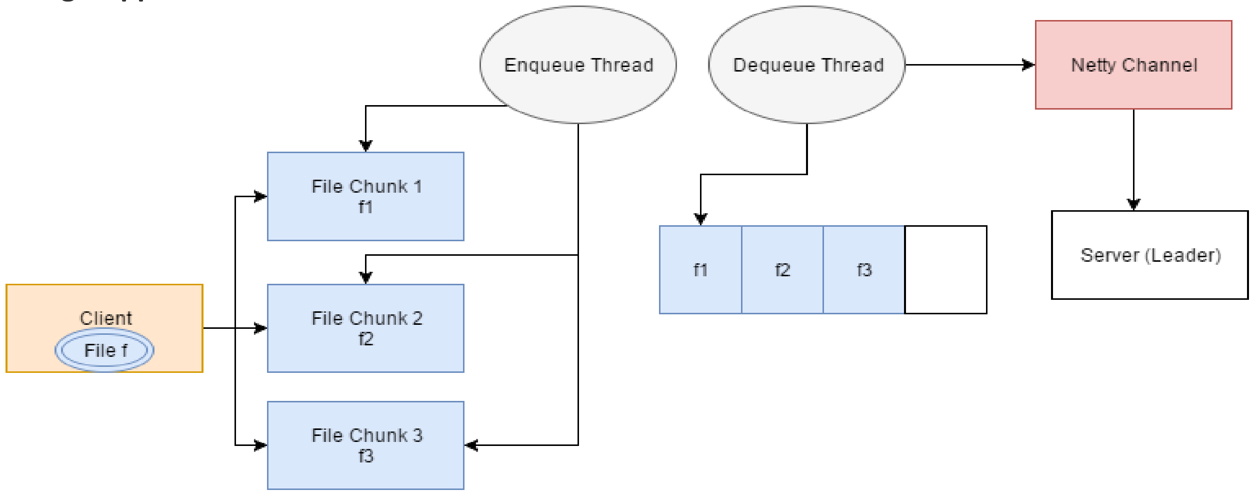


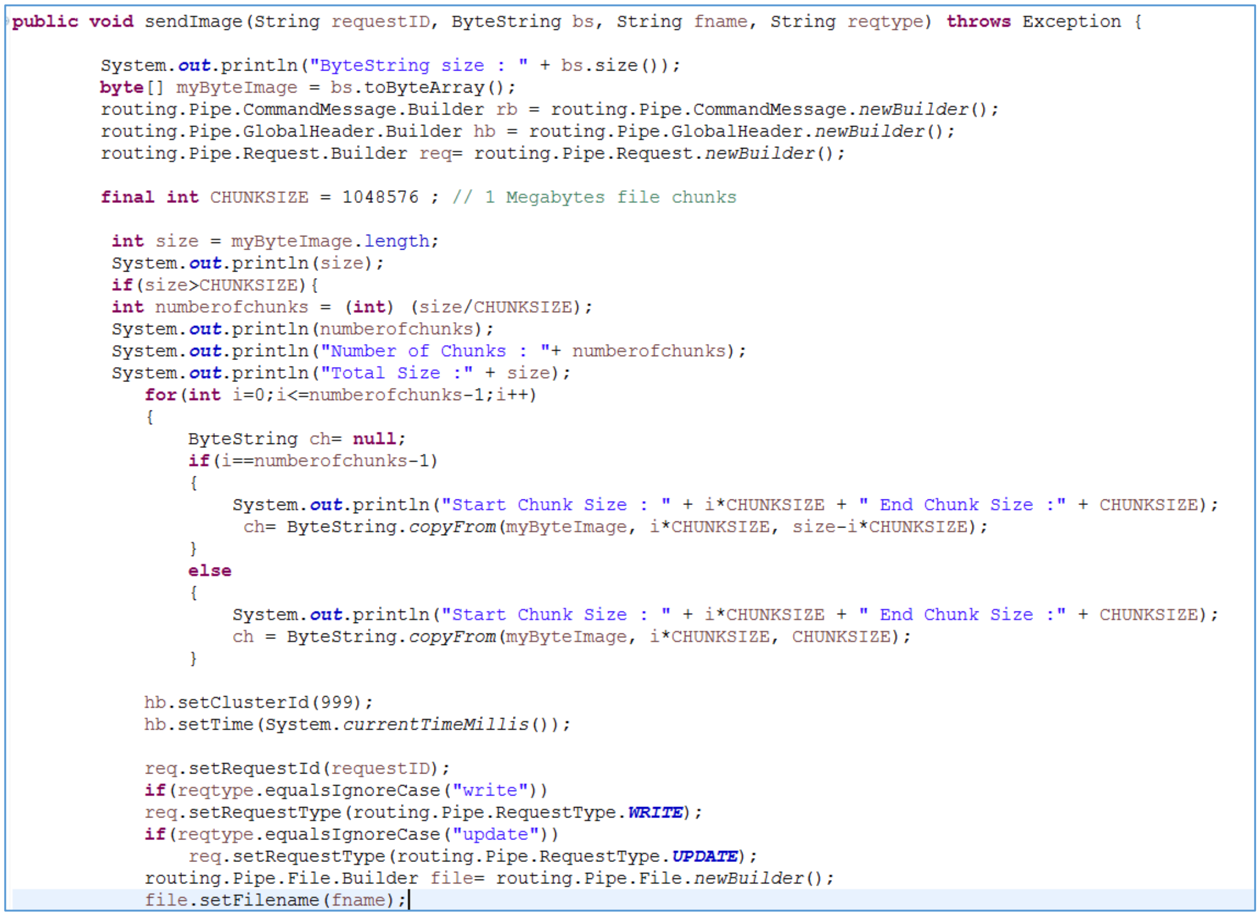
Figure 3: Client-side Architecture for write operation

The above image depicts the client-side architecture and flow of how a file is uploaded in the Distributed database.

Implementation in client-side :

- Initially when the write-operation is initiated, the file size is checked. It it's greater than 1MB, then it is divided into chunks as shown in the snapshot below.

- Each file chunk is enqueued as CommandMessage in a LinkedBlockingQueue as separate write tasks on the client side.
- Another Thread consumes each of these write tasks and pushes them into a netty channel to process in the server side.



```java
public void sendImage(String requestID, ByteString bs, String fname, String reqtype) throws Exception {

        System.out.println("ByteString size : " + bs.size());
        byte[] myByteImage = bs.toByteArray();
        routing.Pipe.CommandMessage.Builder rb = routing.Pipe.CommandMessage.newBuilder();
        routing.Pipe.GlobalHeader.Builder hb = routing.Pipe.GlobalHeader.newBuilder();
        routing.Pipe.Request.Builder req= routing.Pipe.Request.newBuilder();

        final int CHUNKSIZE = 1048576 ; // 1 Megabytes file chunks

         int size = myByteImage.length;
         System.out.println(size);
         if(size>CHUNKSIZE){
         int numberofchunks = (int)  (size/CHUNKSIZE);
         System.out.println(numberofchunks);
         System.out.println("Number of Chunks : "+ numberofchunks);
         System.out.println("Total Size :" + size);
            for(int i=0;i<=numberofchunks-1;i++)
            {
                ByteString ch= null;
                if(i==numberofchunks-1)
                {
                    System.out.println("Start Chunk Size : " + i*CHUNKSIZE + " End Chunk Size :" + CHUNKSIZE);
                     ch= ByteString.copyFrom(myByteImage, i*CHUNKSIZE, size-i*CHUNKSIZE);
                }
                else
                {
                    System.out.println("Start Chunk Size : " + i*CHUNKSIZE + " End Chunk Size :" + CHUNKSIZE);
                    ch = ByteString.copyFrom(myByteImage, i*CHUNKSIZE, CHUNKSIZE);
                }

            hb.setClusterId(999);
            hb.setTime(System.currentTimeMillis());

            req.setRequestId(requestID);
            if(reqtype.equalsIgnoreCase("write"))
            req.setRequestType(routing.Pipe.RequestType.WRITE);
            if(reqtype.equalsIgnoreCase("update"))
                req.setRequestType(routing.Pipe.RequestType.UPDATE);
            routing.Pipe.File.Builder file= routing.Pipe.File.newBuilder();
            file.setFilename(fname);
```

Snapshot: File-chunking and sending method in MessageServer.java

Implementation in server-side:

- Leader receives the file chunks *(as show in Figure 4)* and persisted into its local Database (MongoDB).
- We designed in such a way that a node will have either all chunks of a file or no chunk at all from the file.
- Once the data is persisted in the Leader, it creates a channel will all other active nodes and persists file chunks on the follower's local MongoDB
- Followers sends the acknowledgment on receiving/persisting all the chunks of file to the leader.
- On getting the acknowledgement from majority followers, the leader sends the success response to the client. In-case, leader does not get the acknowledgement from majority followers, it retries for particular time timestamp

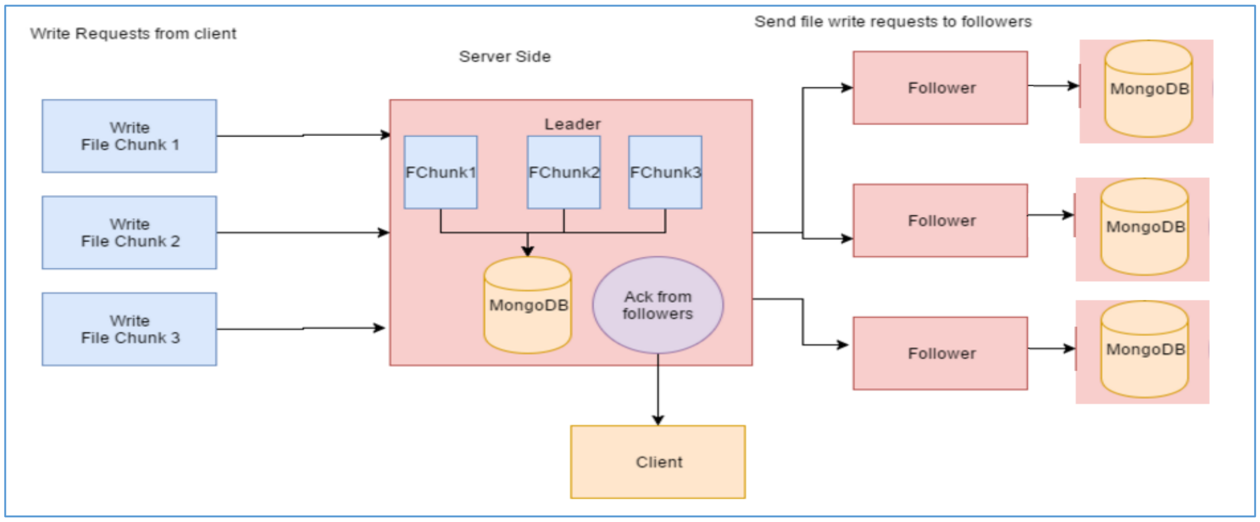
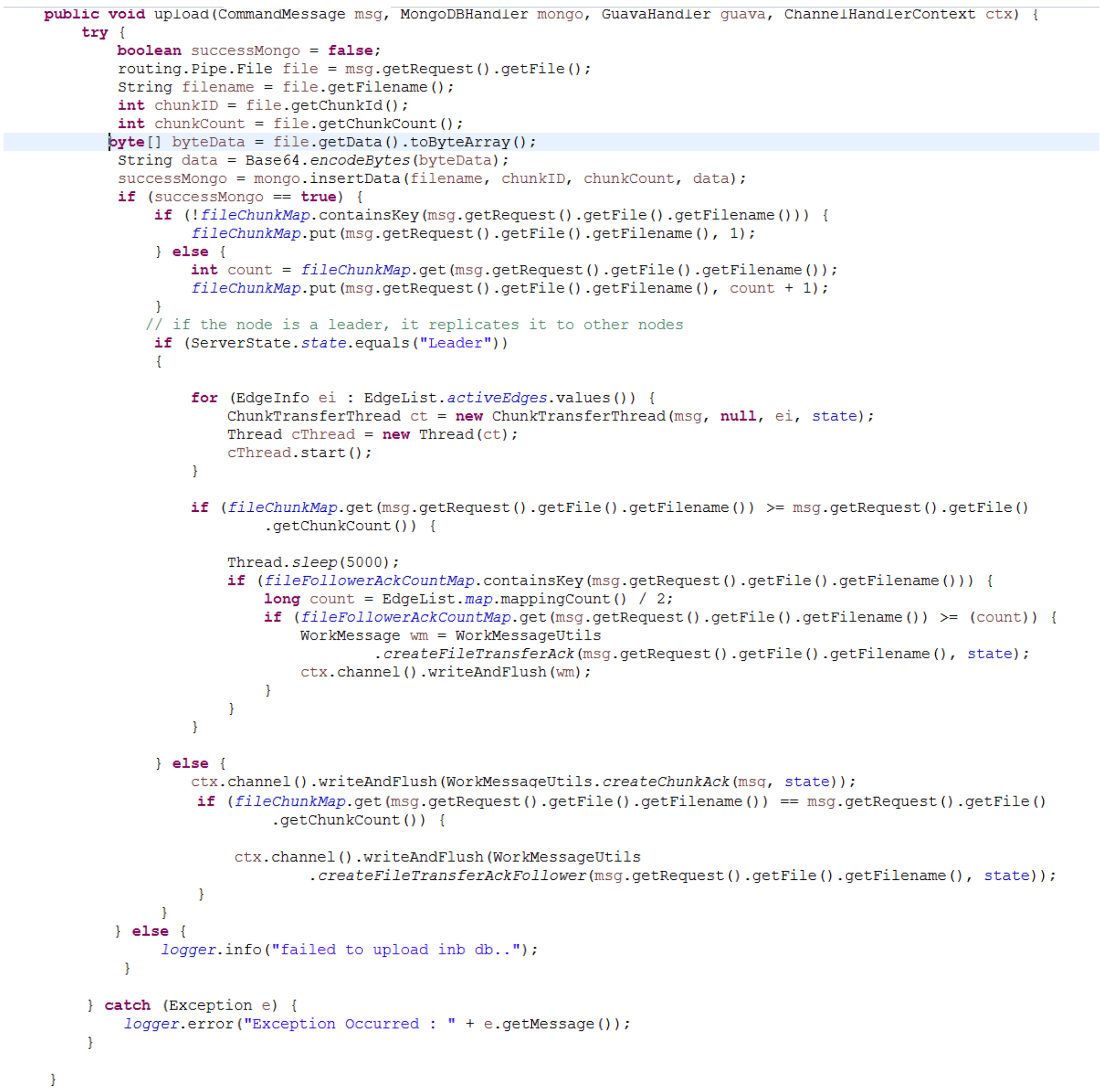


Figure 4: Server Side Architecture for write-operation

Assumption: Majority of the nodes are always up and running and data(file-chunks) for a particular file is always replicated in majority of the followers (all the followers that are alive).

Defense of Design:
1. We have done 100% replication across all the active nodes, inside our cluster in-order to facilitate fault tolerance and 100% availibility.
2. We have designed in such a way that the follower will either get all the chunks of particular file or not even a single chunk. For assuring replication of all chunks on majority active follower or the node which disconnected/connected, we used Acknowledgement from each node to the leader.

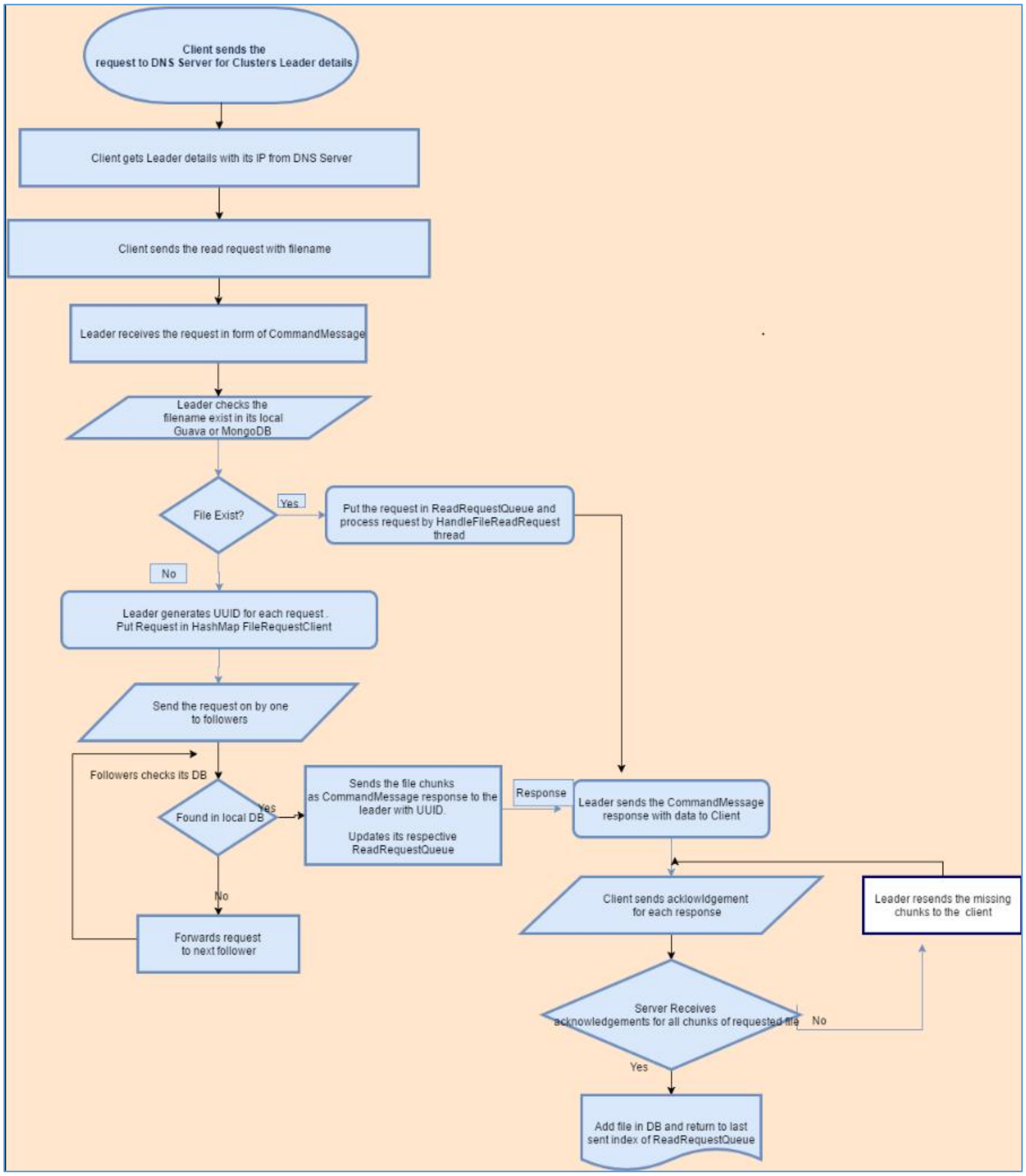


```
    public void upload(CommandMessage msg, MongoDBHandler mongo, GuavaHandler guava, ChannelHandlerContext ctx) {
        try {
            boolean successMongo = false;
            routing.Pipe.File file = msg.getRequest().getFile();
            String filename = file.getFilename();
            int chunkID = file.getChunkId();
            int chunkCount = file.getChunkCount();
            byte[] byteData = file.getData().toByteArray();
            String data = Base64.encodeBytes(byteData);
            successMongo = mongo.insertData(filename, chunkID, chunkCount, data);
            if (successMongo == true) {
                if (!fileChunkMap.containsKey(msg.getRequest().getFile().getFilename())) {
                    fileChunkMap.put(msg.getRequest().getFile().getFilename(), 1);
                } else {
                    int count = fileChunkMap.get(msg.getRequest().getFile().getFilename());
                    fileChunkMap.put(msg.getRequest().getFile().getFilename(), count + 1);
                }
               // if the node is a leader, it replicates it to other nodes
                if (ServerState.state.equals("Leader"))
                {

                    for (EdgeInfo ei : EdgeList.activeEdges.values()) {
                        ChunkTransferThread ct = new ChunkTransferThread(msg, null, ei, state);
                        Thread cThread = new Thread(ct);
                        cThread.start();
                    }

                    if (fileChunkMap.get(msg.getRequest().getFile().getFilename()) >= msg.getRequest().getFile()
                            .getChunkCount()) {

                        Thread.sleep(5000);
                        if (fileFollowerAckCountMap.containsKey(msg.getRequest().getFile().getFilename())) {
                            long count = EdgeList.map.mappingCount() / 2;
                            if (fileFollowerAckCountMap.get(msg.getRequest().getFile().getFilename()) >= (count)) {
                                WorkMessage wm = WorkMessageUtils
                                        .createFileTransferAck(msg.getRequest().getFile().getFilename(), state);
                                ctx.channel().writeAndFlush(wm);
                            }
                        }
                    }

                } else {
                    ctx.channel().writeAndFlush(WorkMessageUtils.createChunkAck(msg, state));
                     if (fileChunkMap.get(msg.getRequest().getFile().getFilename()) == msg.getRequest().getFile()
                            .getChunkCount()) {

                         ctx.channel().writeAndFlush(WorkMessageUtils
                                .createFileTransferAckFollower(msg.getRequest().getFile().getFilename(), state));
                     }
                 }
            } else {
                 logger.info("failed to upload inb db..");
             }

        } catch (Exception e) {
             logger.error("Exception Occurred : " + e.getMessage());
        }

    }
```

Snapshot: Write Operation in MessageServer.java

1.2 <u>File Read Operation</u>

The client-side architecture to download/read a file is similar to the one for the write operation. Only difference is that the files are received in chunks and they are merged at the client side. The flowchart of the process is shown in Figure 5.

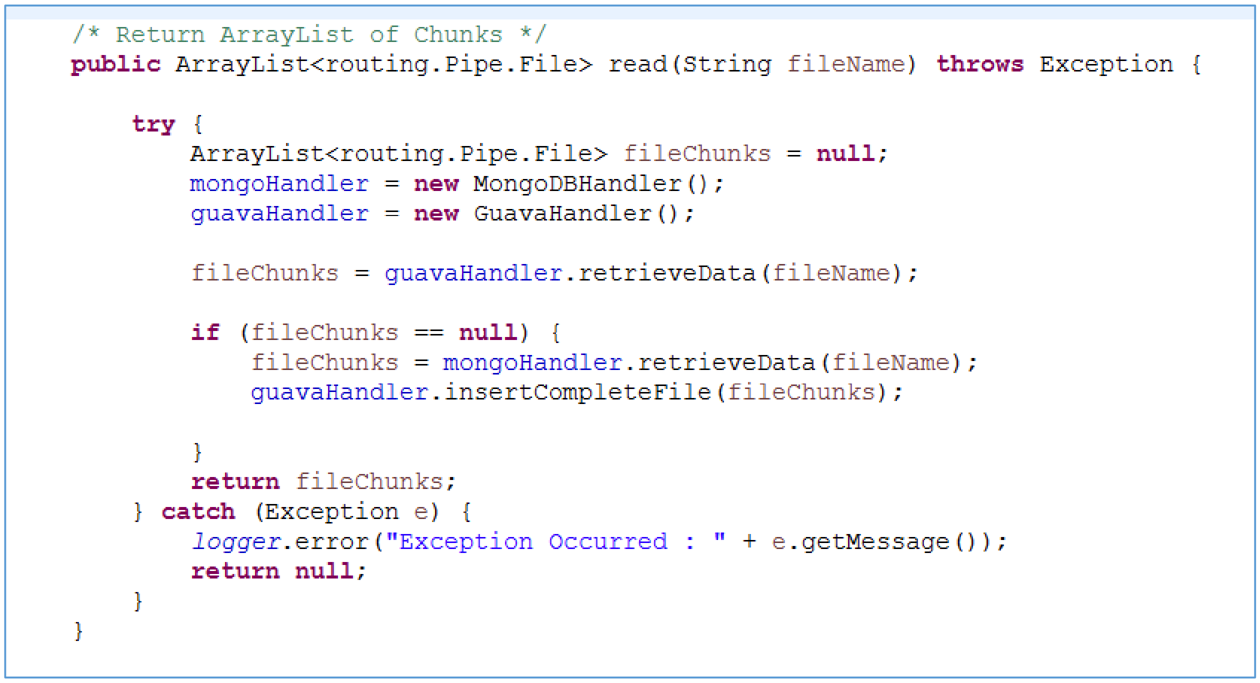


Client sends the request to DNS Server for Clusters Leader details
Client gets Leader details with its IP from DNS Server
Client sends the read request with filename
Leader receives the request in form of CommandMessage
Leader checks the filename exist in its local Guava or MongoDB
File Exist?
Yes
Put the request in ReadRequestQueue and process request by HandleFileReadRequest thread
No
Leader generates UUID for each request . Put Request in HashMap FileRequestClient
Send the request on by one to followers
Followers checks its DB
Found in local DB
Yes
Sends the file chunks as CommandMessage response to the leader with UUID. Updates its respective ReadRequestQueue
Response
Leader sends the CommandMessage response with data to Client
No
Forwards request to next follower
Client sends acklowldgement for each response
Leader resends the missing chunks to the client
Server Receives acknowledgements for all chunks of requested file
No
Yes
Add file in DB and return to last sent index of ReadRequestQueue

Figure 5: Workflow for Read operation

Design Approach

- The leader processes the read request in the form of a FileReadRequest object. A FileReadRequestQueue is used to improve scalability, in case many read requests enter the leader.
- A producer thread enqueues the request into the Request Queue and a consumer thread keeps on reading the request.
- The node (i.e. Leader) tries to fetch requested data from in-memory DB first, if not present fetches data from Mongo and pushed data into the Guava.
- If some chunks are missing, a channel with the followers are reused to fetch chunks.
- The leader consolidates all the chunks and once all the chunks are received, then each of them is sent across to the client.
- Client sends acknowledgement for each response i.e. file chunk. Once server receives acknowledgement of all the file chunks of particular requested file, it starts to process is next request. If not, then server tries to send the missing file chunks again to client.
- The client consolidates all the file chunks and merges them into a single file.



```java
/* Return ArrayList of Chunks */
public ArrayList<routing.Pipe.File> read(String fileName) throws Exception {

    try {
        ArrayList<routing.Pipe.File> fileChunks = null;
        mongoHandler = new MongoDBHandler();
        guavaHandler = new GuavaHandler();

        fileChunks = guavaHandler.retrieveData(fileName);

        if (fileChunks == null) {
            fileChunks = mongoHandler.retrieveData(fileName);
            guavaHandler.insertCompleteFile(fileChunks);

        }
        return fileChunks;
    } catch (Exception e) {
        logger.error("Exception Occurred : " + e.getMessage());
        return null;
    }
}
```

Snapshot : Read Operation in MessageServer.java

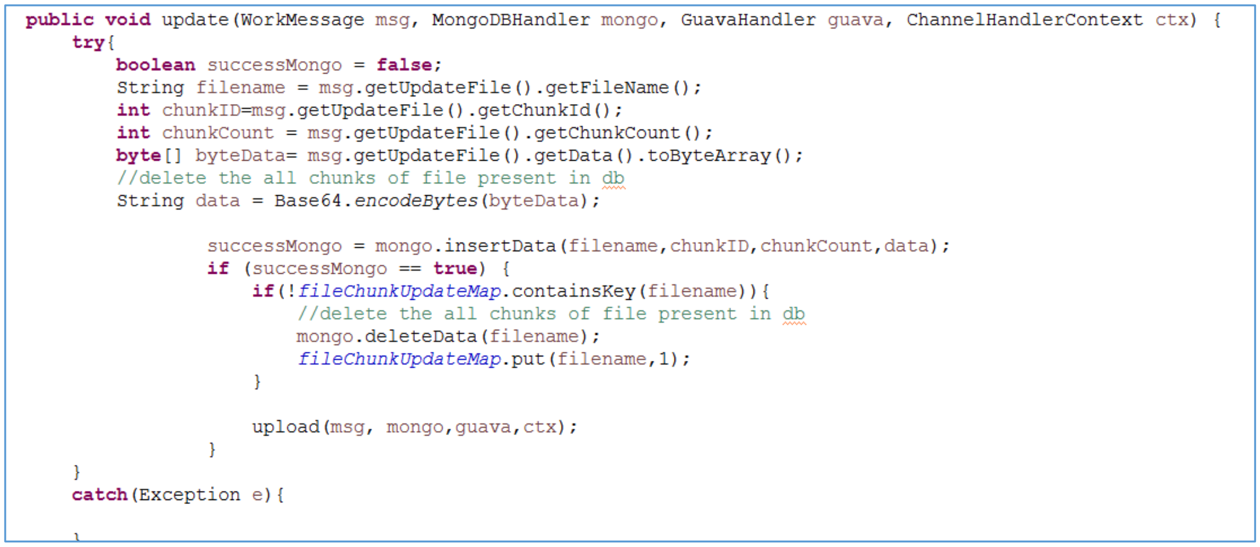
Defense of design

1. For read-operation we are checking first In-memory cache so that that can facilitate faster read operation of recently read files. In-case file is not found in In-memory cache, we are saving file-chunks in Guava (in-memory) after fetching from MongoDB. In Guava, we are using LRU caching algorithm to keep the cache memory updated with recent files (that are read) and when the cache is in full capacity, the oldest file chunks are removed from cache.

1.3 File Update Operation

Design Approach:

1. Design approach for update is similar to the write operation.
2. When leader gets the request for update, it checks if the filename exist in *FileChunkUpdateMap*. If not found, then it first deletes all the files from its Local MongoDB and Guava-cache. Also, it broadcasts the deletion of that filename to all other followers. If filename already exists, then leader directly saves the file chunk in its local MongoDB and also sends that file chunks to all other nodes for update operation.
3. Once file received count which is maintained in *fileChunkMap* for each file name, equals to the totalchunk of that file, then will remove filename from FileChunkUpdateMap, so that we can process request of update for same filename hereafter.



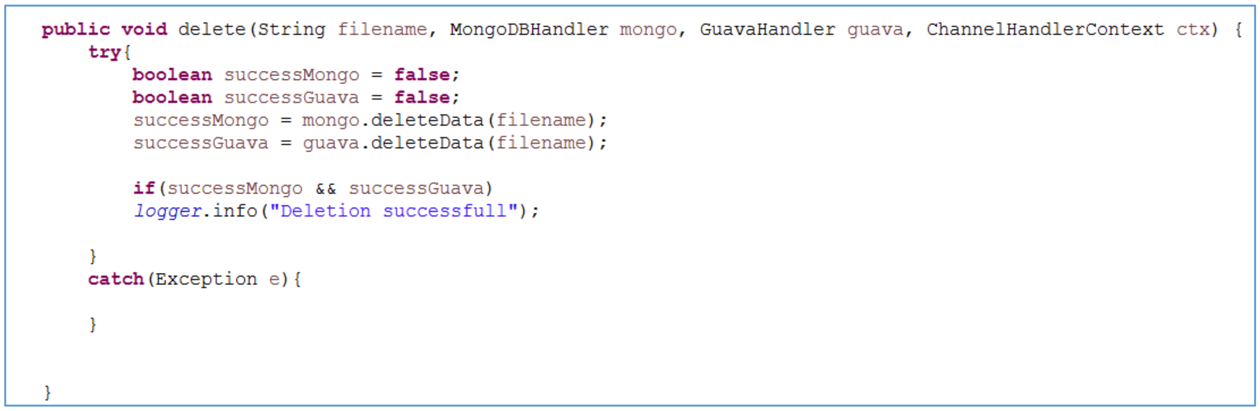
```
public void update(WorkMessage msg, MongoDBHandler mongo, GuavaHandler guava, ChannelHandlerContext ctx) {
    try{
        boolean successMongo = false;
        String filename = msg.getUpdateFile().getFileName();
        int chunkID=msg.getUpdateFile().getChunkId();
        int chunkCount = msg.getUpdateFile().getChunkCount();
        byte[] byteData= msg.getUpdateFile().getData().toByteArray();
        //delete the all chunks of file present in db
        String data = Base64.encodeBytes(byteData);

                successMongo = mongo.insertData(filename,chunkID,chunkCount,data);
                if (successMongo == true) {
                    if(!fileChunkUpdateMap.containsKey(filename)){
                        //delete the all chunks of file present in db
                        mongo.deleteData(filename);
                        fileChunkUpdateMap.put(filename,1);
                    }

                    upload(msg, mongo,guava,ctx);
                }
    }
    catch(Exception e){
```

Snapshot: Update operation in MessageServer.java

1.4 File Delete Operation

Design Approach:

1. For delete requested, the leader calls the DBHandler with Filename to remove the chunk from MongoDB as well as Guava cache



```
public void delete(String filename, MongoDBHandler mongo, GuavaHandler guava, ChannelHandlerContext ctx) {
    try{
        boolean successMongo = false;
        boolean successGuava = false;
        successMongo = mongo.deleteData(filename);
        successGuava = guava.deleteData(filename);

        if(successMongo && successGuava)
        logger.info("Deletion successfull");

    }
    catch(Exception e){

    }


}
```

1.5 Implementation of Leader Election

Each Node can be in one of the three states:

1. Leader
2. Follower
3. Candidate

When each of the nodes/servers start in the cluster, one of them has to claim responsibility as a leader. Hence from a state where none of the nodes are active, to when each of the node of the cluster wakes up, an initial leader election starts and a leader is decided. An election term (the current one) is maintained across all nodes and the same is used to determine the leader.

After this, a leader election occurs only when a leader is down. Leader sends heartbeats(HB) to all followers to let them know that it is alive. When a follower detects that a HB is not received for a given timeout period (randomized for each follower 150-200 ms), the follower becomes a candidate and leader election starts with term one greater than the current.

Once the leader election starts the other followers send votes based on the current term in each follower. If the candidate receives majority votes it becomes the leader and the term is updated in all followers. Another follower becomes a candidate in the next closest time-out period and starts election. In case of split votes, we elect the candidate as a leader considering it as a majority.

Technical Details: ElectionUtil.java is the class that has all the utility methods responsible for leader election algorithm.

Methods Used:

public static void startLeaderElection() ;
Starts the leader election process

public static void broadCastVote(ConcurrentHashMap<Integer, EdgeInfo> map)
Broadcasts Vote to the Candidate

public int getRandomNum(int max, int min);
To get randomized time-out for starting election

public static void createVoteResponse(VoteRequest voteRequest, EdgeInfo ei);
Creates a VoteResponse protobuf message

public static void sendResponse(WorkMessage msg, EdgeInfo ei);
Used to send the vote response

public class BroadCastVote implements Runnable
A Thread class to broadcast vote on a separate thread during election

Defense of Design:

The candidate will request for votes by including its term id in the vote request. A node will vote yes only ifs term number is lesser than that of its own term number. After a leader is elected it will broadcast its new term number to all the nodes. This ensure that the old nodes i.e. nodes with the most complete data will have a higher probability of winning the election if a leader goes down. A node that isn't connected to the leader will not win an election even it starts one since it's term-id will be lesser.

1.6 Implementation Work Stealing

Work Stealing is implemented by the follower. The process is as follows:

1. Follower sends the HeartBeat to other follower to get the queue size.
2. Work stealing request will be successful if the queue i.e ReadRequestQueue size is greater than the threshold value.
3. The leader delegates work if the work stealing successful.
4. Work Stealing thread runs in each of the followers and a request is sent periodically to check the leader's request queue status.

Defense of Design:

1. Leader will not initiate the Work Stealing request since it is assumed that the leader will have its work cut-out and also all file read requests will be sent to leader before it is forwarded to the client. If the work was stolen from the follower by the leader, it will make the process very redundant since the file will have to be sent back to the leader by the follower, creating unnecessary network traffic.
2. Work will be stolen from a node only if the load faced by the node is too high. This is enforced by checking if the number of requests in the queue exceeds a threshold before responding to the workstealing request. If it is lesser than the threshold, then the request is ignored. This method ensures that only the really busy and deserving nodes get the work stolen from them, thus reducing the network traffic significantly.

1.7 Implementation Work Sharing

In this project architecture, the read requests are shared by the leader if the ReadRequestQueue is full. This is a simple Work Sharing strategy that is implemented here.

Defense of Design:

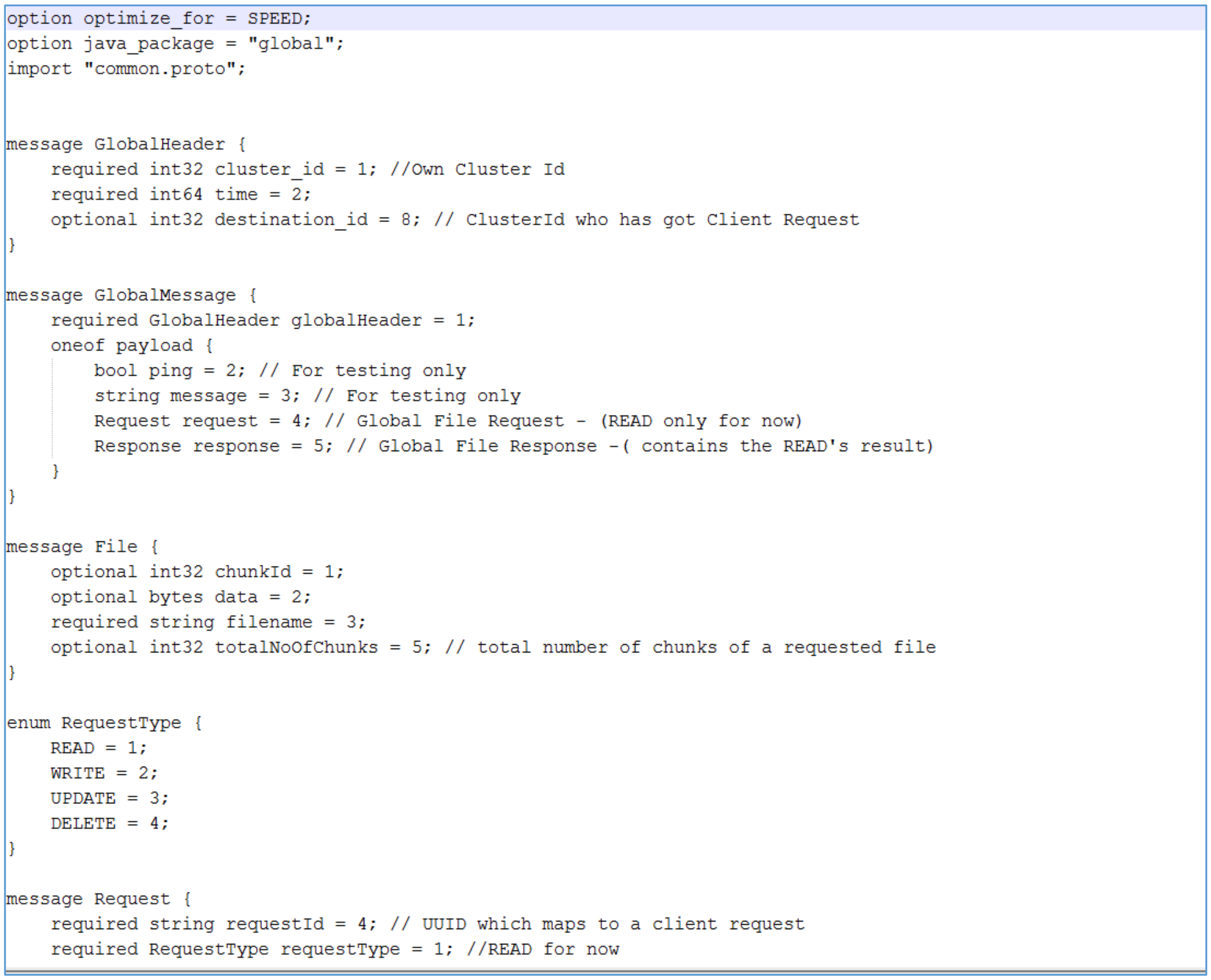
Leader will share its work with a follower only if it is too busy or if the file is not found on its local database i.e. the number of requests in the queue exceeds a threshold. This strategy ensures that , leader will handle requests on its own if the file is present in its database and if it's sufficiently idle.

2. Inter-Cluster Global Communication:

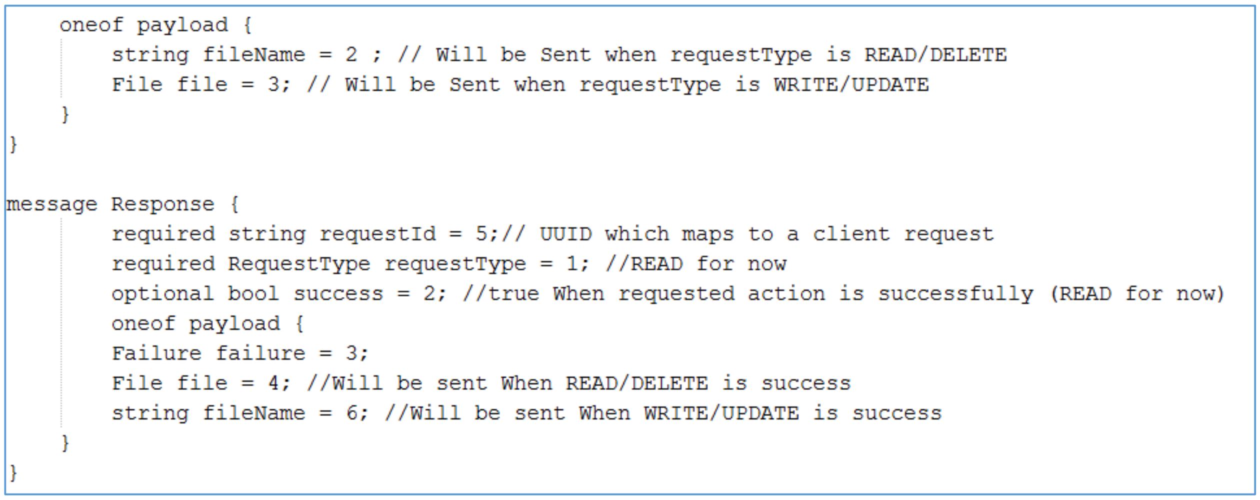
For Inter-cluster communication, we are using GlobalMessage. The inter-cluster communication in the application is processing as shown in the Figure 6.

Design Approach:

1. We used Ring Topology for inter-cluster communication. The globalconfig file is used to map with the node in neighbor cluster.
2. We used Ring Topology because of following reason:
    a) In ring topology all the traffic flows in only one direction at very high speed.
    b) Even when the load on the network increases, its performance is better than that of other topology .
    c) Its high availability n scalability
    d) it is the simplest topology
3. Bidirectional communication would be better option than ring topology because with ring topology
    a) Each packet of data must pass through all the computers between source and destination. This makes it slower than Star topology or other bi-directional topology
    b) one workstation or port goes down, the entire network was getting affected



```
option optimize_for = SPEED;
option java_package = "global";
import "common.proto";


message GlobalHeader {
    required int32 cluster_id = 1; //Own Cluster Id
    required int64 time = 2;
    optional int32 destination_id = 8; // ClusterId who has got Client Request
}

message GlobalMessage {
    required GlobalHeader globalHeader = 1;
    oneof payload {
        bool ping = 2; // For testing only
        string message = 3; // For testing only
        Request request = 4; // Global File Request - (READ only for now)
        Response response = 5; // Global File Response -( contains the READ's result)
    }
}

message File {
    optional int32 chunkId = 1;
    optional bytes data = 2;
    required string filename = 3;
    optional int32 totalNoOfChunks = 5; // total number of chunks of a requested file
}

enum RequestType {
    READ = 1;
    WRITE = 2;
    UPDATE = 3;
    DELETE = 4;
}

message Request {
    required string requestId = 4; // UUID which maps to a client request
    required RequestType requestType = 1; //READ for now
```



```
    oneof payload {
        string fileName = 2 ; // Will be Sent when requestType is READ/DELETE
        File file = 3; // Will be Sent when requestType is WRITE/UPDATE
    }
}

message Response {
        required string requestId = 5;// UUID which maps to a client request
        required RequestType requestType = 1; //READ for now
        optional bool success = 2; //true When requested action is successfully (READ for now)
        oneof payload {
        Failure failure = 3;
        File file = 4; //Will be sent When READ/DELETE is success
        string fileName = 6; //Will be sent When WRITE/UPDATE is success
    }
}
```

Snapshot: GlobalMessage in global.proto

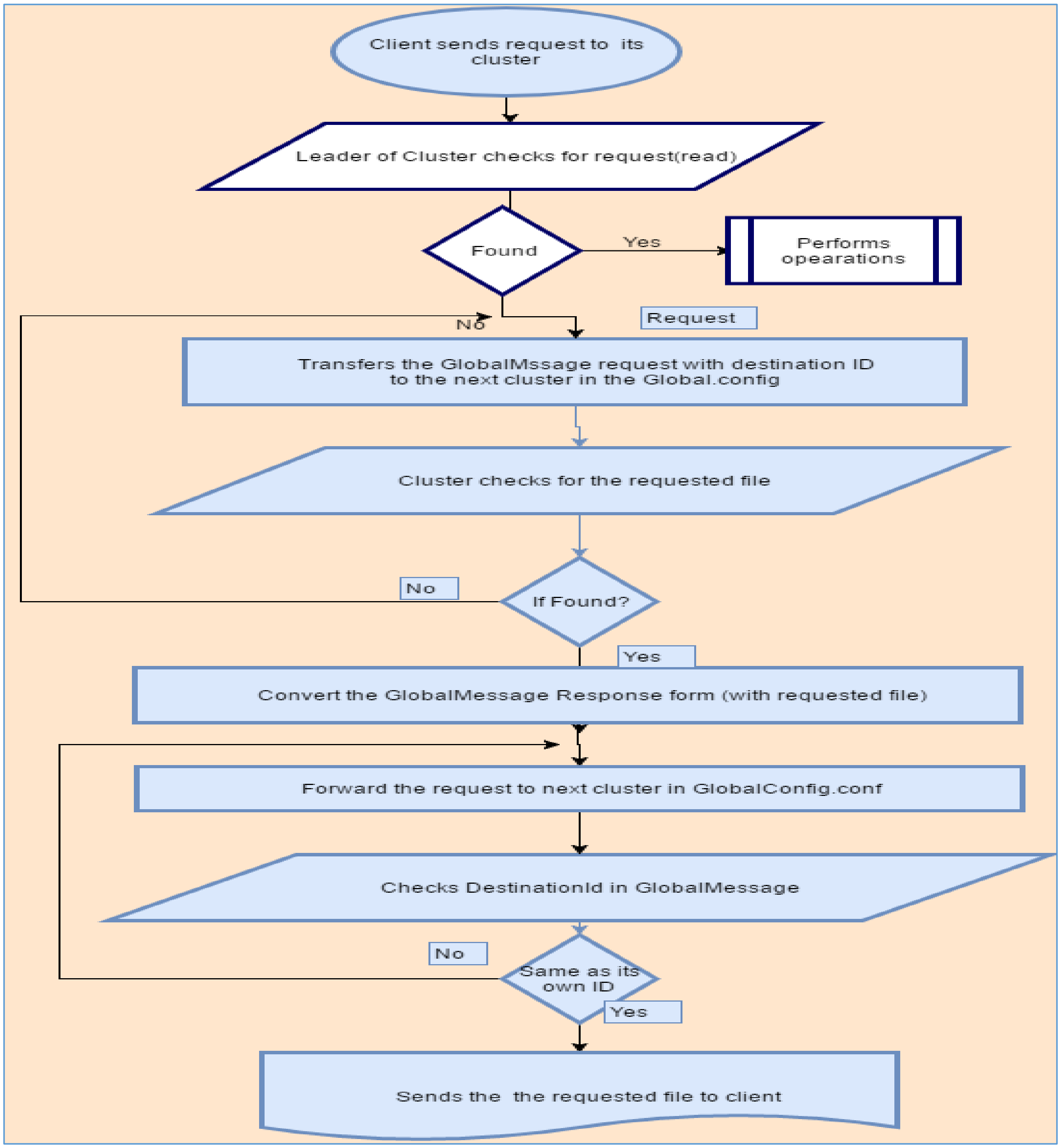

Client sends request to its cluster
Leader of Cluster checks for request(read)
Found
Yes
Performs opearations
No
Request
Transfers the GlobalMssage request with destination ID to the next cluster in the Global.config
Cluster checks for the requested file
No
If Found?
Yes
Convert the GlobalMessage Response form (with requested file)
Forward the request to next cluster in GlobalConfig.conf
Checks DestinationId in GlobalMessage
No
Same as its own ID
Yes
Sends the the requested file to client

Figure 6 : Workflow of Inter-Cluster global communication

Problems Faced during Inter-Cluster Global Communication Integration and Testing:

1. The Chunk id’s of files were indexed with different base values 0 or 1 by different teams. Due to this there was an issue with retrieval of file as the teams used chunk id to retrieve chunks. Finally we agreed upon a Zero based index as majority teams were using it.
2. The time taken for the teams to co-ordinate and get the test cases done was too long, as this was proportional to the number of teams.
3. The naming convention of destination id is misleading and it refers to the first cluster id in the distributed system (ideally should have been the source id).
4. This id that has to remain unhindered during the propagation of global message , but was set by some teams to their own cluster id. Hence we had to co-ordinate with those teams and get them to modify their code.

5. One of the teams used a queue to get all incoming global request and handled them in a single thread which caused a significant latency in the ring when large files requests were transferred.

Testing and Benchmarking

IntraCluster Testing :

Request Type	Number of requests	Size	Time Taken	Test ClassName
Write	1	< 1MB	193 ms	OneWriteRequest
Write	1	<10 MB	1350 ms	MediumSizeWriteTest
Write	1	50-60 MB	2881 ms	LargeSizeWriteRequests
Write	10	<1 MB	1535 ms	TenWriteRequests
Write	100	<1 MB	20998 ms	HundredWriteReqs
Read	1	<1 MB	241 ms	OneReadRequest
Read	1	<10 MB	507 ms	MediumSizeReadRequest
Read	1	50-60 MB	927 ms	LargeSizeReadRequest
Read	10	<1 MB	3622 ms	TenReadRequests
Read	10	<10 MB	5017 ms	MediumTenReadRequests
Read	10	50-60 MB	7194 ms	LargeTenReadReqs
Read	100	<1 MB	17226 ms	HundredReadReqs
Read	100	<10 MB	19562 ms	MediumHundredReadReqs
Read	100	50-60 MB	23736 ms	LargeHundredReadReqs

Observations:

1. As the number of requests increase the time taken to read or write a file increases linearly with respect to the number of requests. The reason is that the requests are queued at the client side and only one CommWorker thread retrieves the request from the queue and sends it to the server.
2. Such observation is not found with respect to the size of the request. The reason is that write request of each chunk of the file is handled in a separate thread in the server side.
3. This shortcoming of latency during multiple requests can be avoided at the client side by having number of threads to manage the requests in the Worker Queue.

Contribution

- Balamurugan : *Leader Election, File Handling(client/server), Junit test cases, DNS Server, Global Interface, Report*
- Mukul : *File Handling(client/server), DNS Server, Database(Guava, MongoDB), Python Client, File Chunking, Report*
- Saikrishnan : *Leader Election, Work Stealing/Sharing, Global Interface, Report*

References

https://developers.google.com/protocol-buffers/docs/proto
http://seeallhearall.blogspot.com/2012/05/netty-tutorial-part-1-introduction-to.html
http://netty.io/wiki/user-guide-for-4.x.html
https://www.tutorialspoint.com/python/python_networking.htm
http://zetcode.com/articles/guava/
https://docs.mongodb.com/v3.2/tutorial/