­­­­

***A project report on CMPE 275 Project 1***

**FLUFFY**

(A distributed storage system)

**Team Amigos**:

Akshatha Madapura Anantharamu

Kamlendra Singh Chauhan

Himanshu Jain

Savio Fernandes

Pramod Kasibhatta

**Contents**

[1. Introduction 4](#_Toc448005398)

[2. Technologies Used 4](#_Toc448005399)

[1.1 Netty 4](#_Toc448005400)

[Features of Netty: 4](#_Toc448005401)

[1.2 Google Proto Buffer 5](#_Toc448005402)

[Protobuf vs XML: 6](#_Toc448005403)

[Protobuf vs JSON: 6](#_Toc448005404)

[1.3 Apache ANT 6](#_Toc448005405)

[Features of ANT: 6](#_Toc448005406)

[Advantages of ANT: 6](#_Toc448005407)

[3. Topology 7](#_Toc448005408)

[Advantages of Mesh Topology: 7](#_Toc448005409)

[Disadvantages of Mesh Topology: 7](#_Toc448005410)

[4. Architecture and Workflow 8](#_Toc448005411)

[4.1 WRITE request from client directly connected to cluster 8](#_Toc448005412)

[4.2 READ request from client directly connected to cluster 9](#_Toc448005413)

[4.3 READ request from client outside cluster 10](#_Toc448005414)

[5. Leader Election 11](#_Toc448005415)

[5.1 Raft Leader Election 11](#_Toc448005416)

[Why Raft? 11](#_Toc448005417)

[Implementation of Raft 12](#_Toc448005418)

[6. REPLICATION 17](#_Toc448005419)

[6.1 Data Replication Implementation: 17](#_Toc448005420)

[6.2 Why Replication? 18](#_Toc448005421)

[6.3. RIAK 18](#_Toc448005422)

[Why RIAK: 19](#_Toc448005423)

[6.4. Mongo Data Base 19](#_Toc448005424)

[Advantages of Mongo DB: 19](#_Toc448005425)

[6.5. Auto Replication 20](#_Toc448005426)

[7. Node Discovery 20](#_Toc448005427)

[7.1 Why Node discovery: 20](#_Toc448005428)

[Process 21](#_Toc448005429)

[8. Work Stealing 23](#_Toc448005430)

[9. Queuing 25](#_Toc448005431)

[9.1. Why Queuing? 25](#_Toc448005432)

[9.2 Types of Queues 25](#_Toc448005433)

[9.3 Advantages of Queuing 26](#_Toc448005434)

[9.4 Additional Components 26](#_Toc448005435)

[9.5 Detailed Explanation of the Queuing System 27](#_Toc448005436)

[10. Inter-Cluster Communication 35](#_Toc448005437)

[10.1 Working of Global Command Adapter 36](#_Toc448005438)

[11. Client API 38](#_Toc448005439)

[12. Test Cases 41](#_Toc448005440)

[Learnings 45](#_Toc448005441)

[Future Work 46](#_Toc448005442)

[Conclusion 47](#_Toc448005443)

[References 47](#_Toc448005444)

[Contributions 47](#_Toc448005445)

# Introduction

In day to day life data is being generate in a very huge amount. Data being stored on hardware is not reliable because it can be destroyed which may lead to data loss. To prevent data loss we need a reliable, scalable and efficient storage system. These systems should be able to handle data in huge amounts. Today we are left with choosing one or more existing cloud storage systems such as Dropbox, Google Drive, server installation like HDFS, etc... So we are introducing a new decentralized platform to this list and its being named *“Fluffy”.*

Fluffy is a distributed file storage system built for storing and retrieving the files. Users are allowed to store and retrieve any type of file varying from docx, pdf, zip documents also images of format PNG, JPEG and multimedia file like mp3. It’s a highly scalable, secure, fault tolerant and efficient file storage system able to handle multiple queries simultaneously.

# Technologies Used

Fluffy is being built by in-cooperating the following technologies,

|  |  |
| --- | --- |
| Languages | Java, Python |
| Core Packages | Netty, Google Protobuff |
| Databases | Mongo DB, RIAK |
| Builder | Apache Ant |

Following section is going to give brief introduction about core packages Netty, Google Protobuff and also about builder Apache Ant.

## Netty

Netty is a NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients. It greatly simplifies and streamlines network programming such as TCP and UDP socket server [1].

Netty is a Java library and provides unified Asynchronous API for writing networked and networking applications and services. Same API can be used for different transport types so there will be no change in code with respect to change in transport type. It provides feature to handle multiple connection with single thread which saves lot of memory.

### Features of Netty:

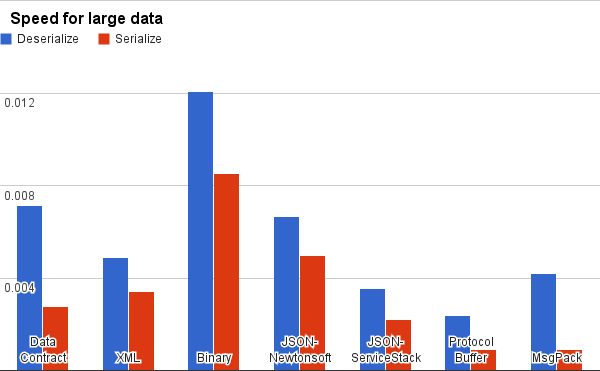
* Unified API for blocking and non-blocking sockets.
* Highly customizable thread model.
* Less resource consumption, better throughput and low latency.
* Minimize unnecessary memory copy.
* Complete SSL/TLS and StartTLS support.
* Flexible and extensible event model.
* Connectionless datagram socket support.

## Google Proto Buffer

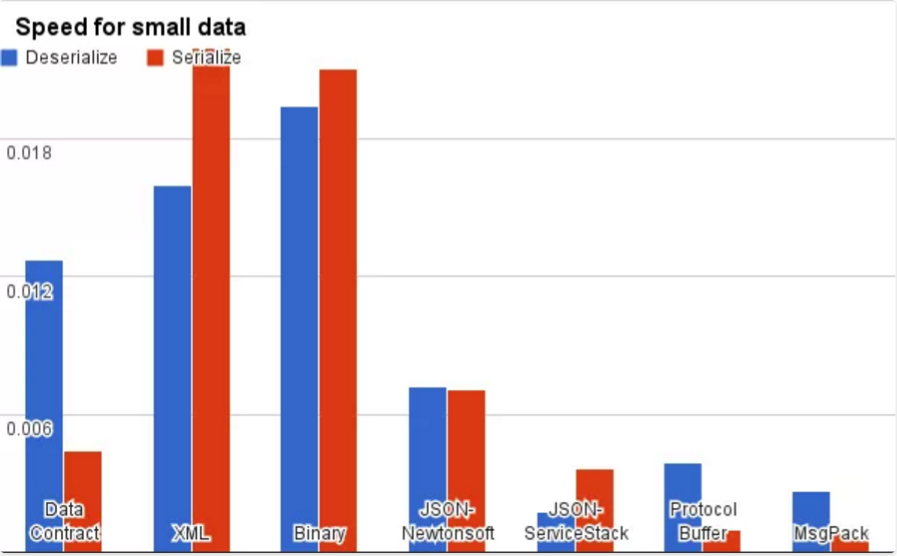
Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages [2].

Proto buffers are way of encoding structured data in efficient yet extensible. Initially we need to define structure of message to be used in proto file. By using proto buffer compiler when we compile these proto files it will generate class that contain methods for each of fields in proto files. These files can be used to automatically encode and decode data.

Below is speed comparison chart for serialization and deserialization of small and large data across various data format [3]. As we can protobuf is extremely fast in both the cases.



**Fig 1: Speed chart for large data**



**Fig 2: Speed chart for small data**

### Protobuf vs XML:

* Protobuf is 20 to 100 times faster than XML.
* Protobuf is 3 to 10 times smaller than XML.
* Protobuf generates classes which can be used to access the data which helps which is easier to use programmatically.
* Protobuf is simple and less ambiguous compared to XML.

### Protobuf vs JSON:

* Protobuf has schema which enforces the structural components of our data which are very important.
* In Protobuf, new fields can easily be introduced as it obviates the need for version check.
* In JSON you have to write code to handle encoding and decoding to and from JSON objects. It also requires us to write code to parse each newly added data type.
* In Protobuf, as we change the schema, the proto generated classes also evolve. We just have to regenerate them.
* The required, optional and repeated keywords in Protobuf are very powerful.

## Apache ANT

It’s a Java based build tool from Apache Software Foundation. It’s a tool for automating build process. It provide number of built in task to allow to compile, assemble, test and run java application. ANT build files are written in XML and are portable, easy to understand and open-source. Its works based on targets and dependencies defined in XML files. It can be also used to build other source code like C, C++.

### Features of ANT:

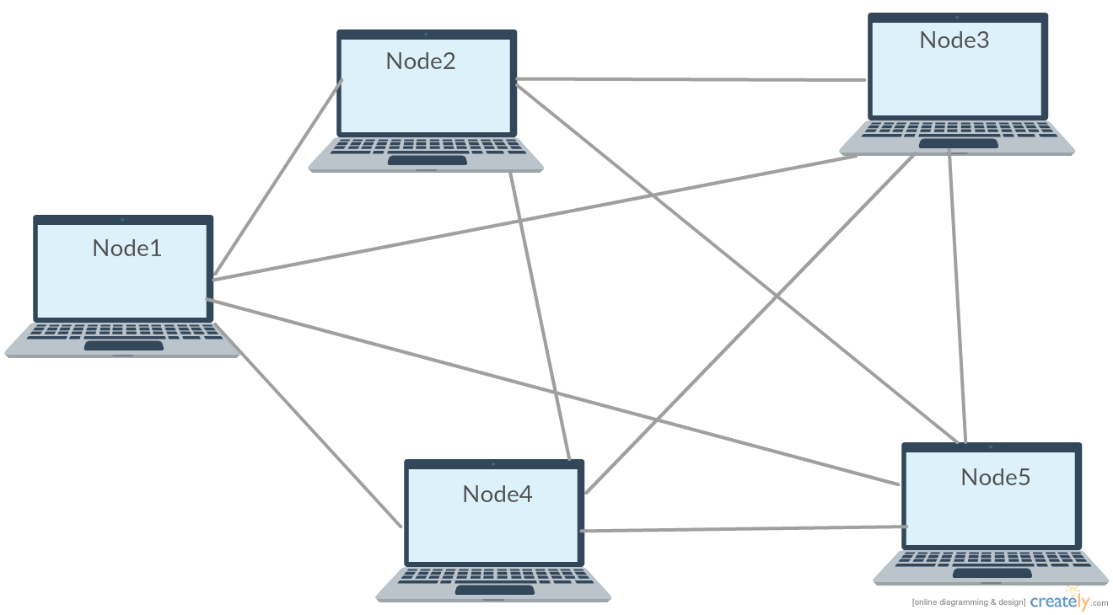
* Compile java based applications and create Java Doc
* Create war, jar, zip, tar files and also directories.
* Can copy files to at different locations and also can delete or move files
* Can send Emails to the stakeholders
* Supports Junit 3, Junit 4, testing etc.
* Convert XML based test reports to HTML reports
* Can check out the code from version control system (SVN, GIT, CVS etc.).
* Can execute test scripts and test suites

### Advantages of ANT:

* Ant provides wide range of task and fulfils almost every requirement of user.
* Ant is platform independent and only requirement is JDK.
* We can extend tool capability by writing java code for adding task into ANT lib.

# Topology

The topology that we have implemented in our project is *“mesh topology”.* Every node in network is interconnected to every other node. Below Fig3 show the mesh topology.



**Fig 3: Mesh Topology**

### Advantages of Mesh Topology:

* Data transmission can be done simultaneously across different nodes.
* Can withstand high traffic.
* In the case of node failure there is always alternative path.
* Addition and deletion of nodes can be done without effecting other nodes.

### Disadvantages of Mesh Topology:

* Setup, Maintenance and administration of network is very difficult.
* Overall network cost is high compared to other networks.
* High chance of redundancy in many of network connections.

# Architecture and Workflow

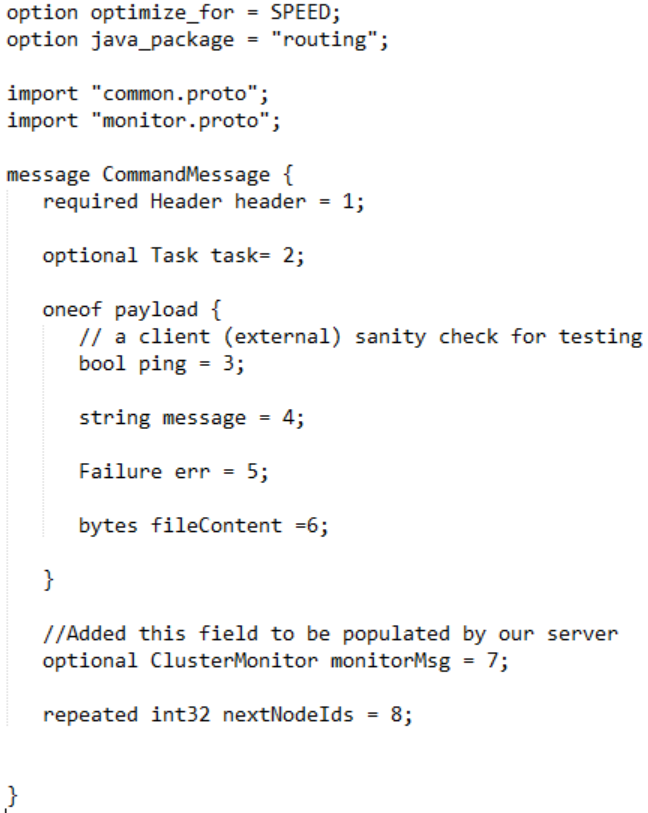
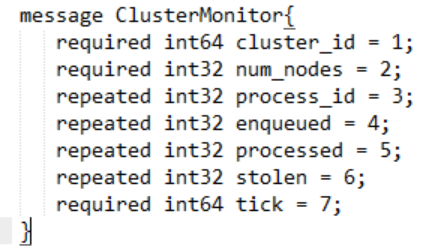
In this section we are going to discuss how a request from client is being handled. There are 3 types of request that can be received and should be processed.

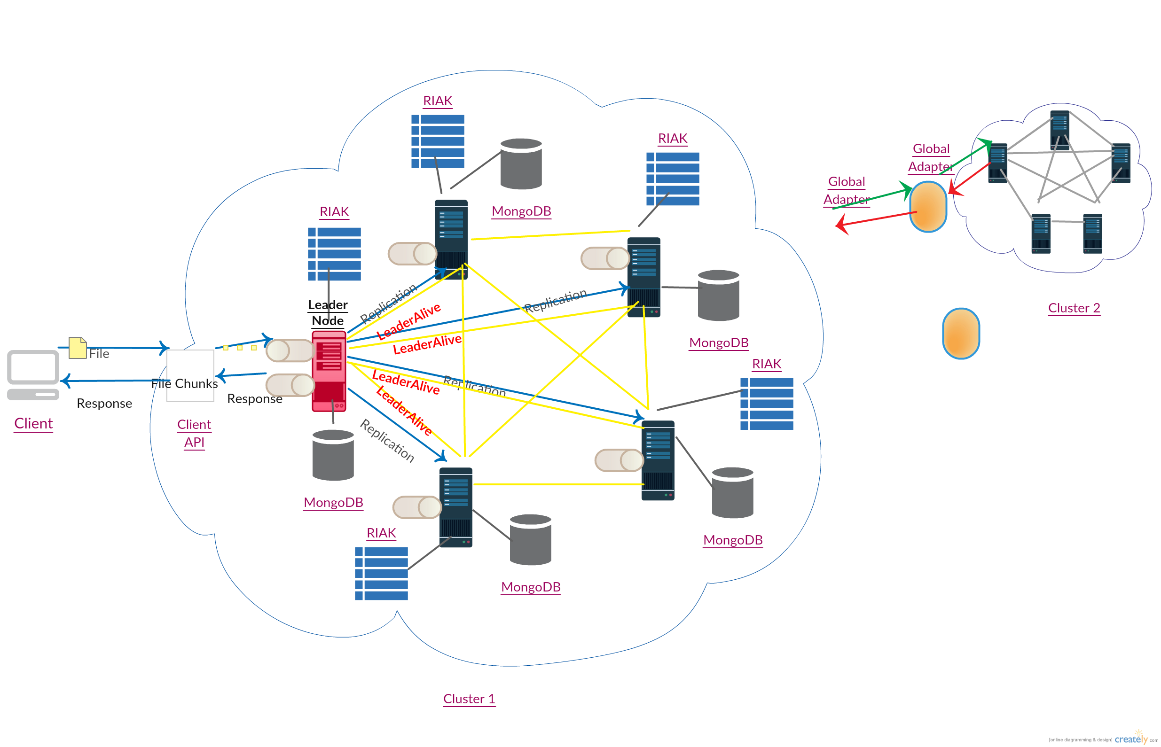
1. WRITE request from client directly connected to cluster
2. READ request from client directly connected to cluster
3. READ request from client outside cluster

## WRITE request from client directly connected to cluster

Client sends a WRITE request, i.e. request to store a file this will be accepted by client API. In client API, file is divided into chunks of 1MB. After file is being divided into chunks a CommandMessage is being created for each chuck along with other necessary information. Below we can see the structure of CommandMessage.

**CommandMessage.proto**

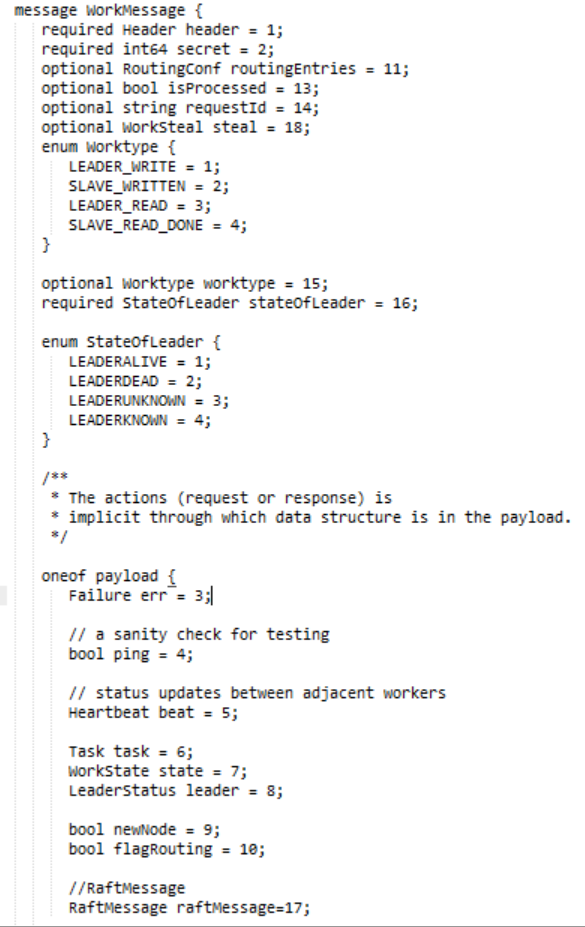


Now we have generated Command Message for each chunk, we are going to send each of these message to Leader Node at its Command Port. These messages will be enqueued into Inbound Queue of Leader Node. Leader Node will process each message one after the other. Firstly it will check if incoming file size is less than 1MB or not by looking into no of chunk’s field. If the file size less than or equal to 1MB then the noofchunks field will be equal to 1. If file is less than 1MB then it will be stored on RIAK database of Leader Node and sent to replication to other RIAK database of follower nodes. Once it receive acknowledgement from majority of followers for successful storage of file chunks then Leader Node is going to send success message back to client. If file is greater than 1MB the process repeats but instead of RIAK database, file chunks will be persisted on Mongo database. Below figure represent the discussed workflow.

**Fig 4: WRITE request from client directly connected to cluster**

## READ request from client directly connected to cluster

Client sends a READ request, i.e. request to retrieve a file this will be accepted by client API. In client API, a CommandMessage is being created for file to be retrieved with necessary information. Now we have generated Command Message, we are going to send this message to Leader Node at its Command Port. These message will be enqueued into Inbound Queue of Leader Node. It will delegate request to one of the client to handle. If file is less than 1MB then it will be stored on RIAK database else it will be found on Mongo database. According to size of file client generate WorkMessage (<1MB)/WorkMessages (>1MB) and forward to Leader Node. Below is the Structure of WorkMessage is sent to client. Suppose a file was not found in the cluster then it will be forwarded to Global Adapter from the Leader Node. This scenario is explained in the next section.

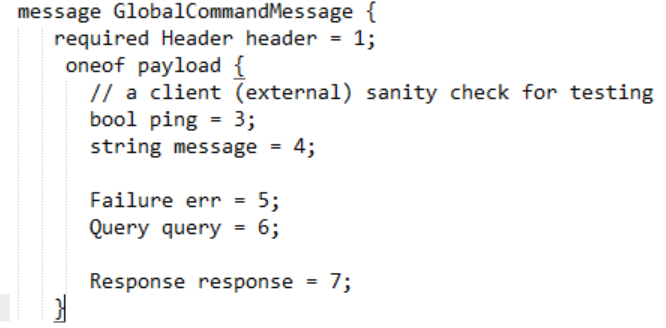




**WorkMessage.proto**

## READ request from client outside cluster

Suppose we receive a READ request from Global Adapter, i.e. client had sent request to retrieve file to a particular cluster but file was not found in that cluster then request was forwarded to its interconnected cluster through Global Adapters of their respective cluster. The message we receive on Global Adapter will be in the GlobalMessage. In Global Adapter GlobalMessage will be converted to CommandMessage and send to the Leader Node of the cluster. Further read request will be processed as described in the previous section and the file chunks is sent to Global Adapter from Leader Node as CommandMessages and further Global Adapter will convert it to GlobalMessages and send to the Global Adapter from which it received request. Then it will again convert to CommandMessage and send to its Leader Node and Leader Node will further forward to client API where chunks are combined and final file is sent to client.



**GlobalMessage.proto**

# Leader Election

In a distributed environment, the process of assigning a node the responsibilities of handling various activities is called Leader election. The activities includes:

* Maintaining the alive status of all the nodes in the cluster (by sending heartbeat messages).
* Delegation of work to other nodes.
* Replicate the incoming data to other nodes to have a redundant data storage environment.

The leader election is a dynamic process and whenever the nodes are unaware of the current leader node, they have the opportunity to start a new election and based on the algorithm we implement, the election happens. For this project we went ahead with RAFT leader election.

## 5.1 Raft Leader Election

Raft uses randomized timers running on all the nodes to start the election. The timer of whichever node gets finished first broadcasts the vote request to all other nodes in the cluster and based on the current state of the other nodes the respond. The leader gets elected if and only if it receives majority of the votes. The algorithm is quite straightforward but very powerful.

### Why Raft?

We are using Mesh topology for our network i.e. all the nodes are connected to all other nodes. Raft is one of the algorithms which considers all the nodes to be connected to all other nodes so that nodes can send all the requests directly to them i.e. vote request, vote response, heartbeats etc. Raft is easy to understand and implement, it’s efficient because there is no need to know about the size of the network. It is powerful for log replication and consensus.

### Implementation of Raft

#### Requirements:

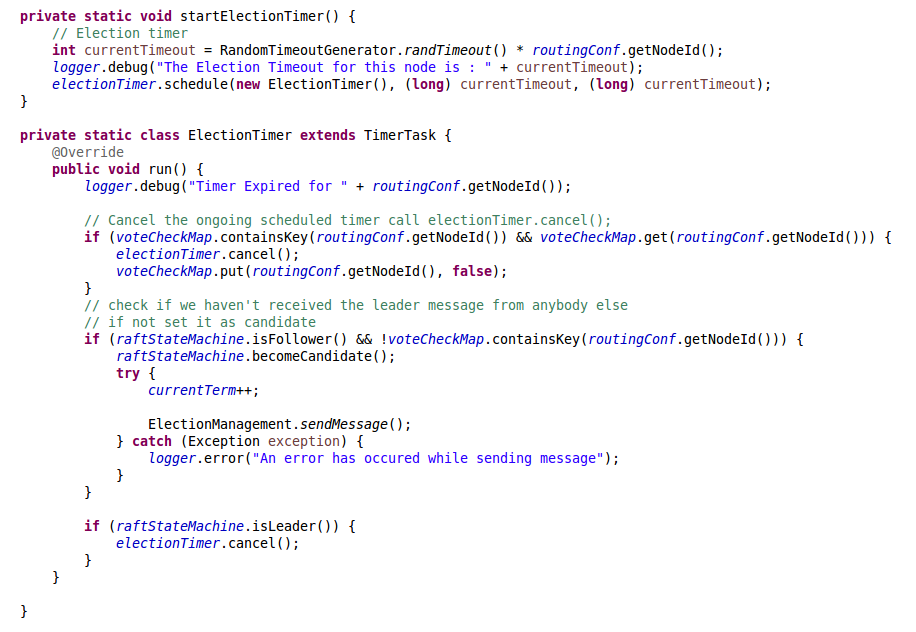
To start the election process there should be at least three nodes in the cluster otherwise election won’t start.

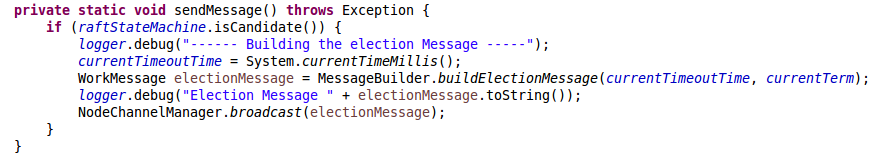
#### Process:

A node can be in any of the three states:

1. Follower
2. Candidate
3. Leader

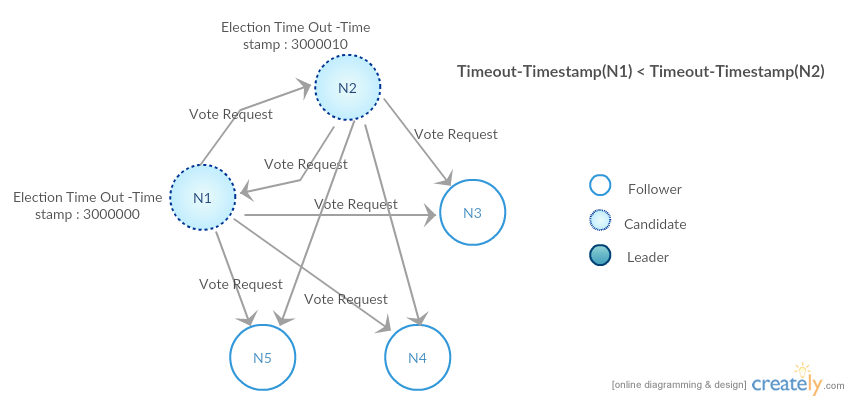
The **RaftNodeStateMachine** maintains the state of a given node. The state machine changes the state whenever different events happen. Initially all the nodes are in the follower state. The election timer (a randomized timer) starts on all the nodes as soon as election gets triggered. The trigger here is the **EdgeMonitor** which continuously checks for the connecting edges and as soon as the number of outer edges becomes two or more, **MessageServer** thread starts election. The node whose election timer gets timed out first, becomes the candidate node and sends out vote request to all other nodes.





Upon receiving the request vote there are various situation which might arise on receiving nodes based on following states of these nodes:

1. **Follower**: If the receiver is in follower state, it simply grants vote to the candidate and update its state i.e. the vote has been given in this term (Explained later). Along with this it immediately stops its timer and waits for leader to get elected.
2. **Candidate**: If the receiver is in candidate state, it means that the timer for this node has timed out before receiving the vote request from another node and now there is conflict. To resolve this situation we have introduced a **Node** **Timeout Timestamp** field which is being added to Vote Request Message, which is being sent out from candidate to other nodes and is maintained as part of the node state. Once Node 1 receives the message, it check for the the timeout value of both the nodes. If the timeout for other node is lesser than its own election timeout, it sends out the vote granted response to other node otherwise vote is not granted to other node.

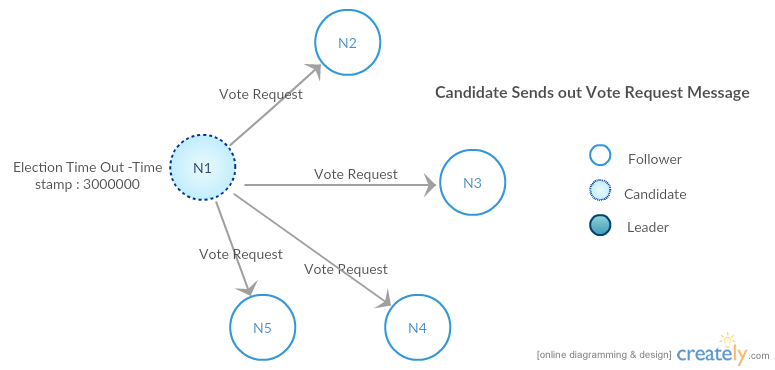


1. **Leader:** If leader gets the election message for any reason, there are two possibilities:

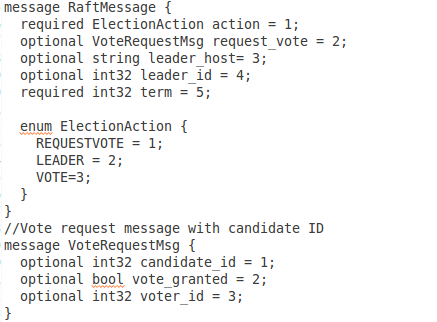
A: If network partition would have happened and the current node got disconnect from the rest of the network. Once the network becomes available again the current node will get message from other other node for leader election and at that time term number should be considered to determine which node should be the ultimate leader of the cluster. Ideally the node with the newer term should be the new leader.

B: The leader sends out the leader alive message to all the nodes in the cluster to let them know that the leader of the cluster is working properly but due to any network related problems it is possible that the alive message got delayed and any of the other node in the cluster changed to the candidate state and triggers leader election. In such situations re-election will happen.

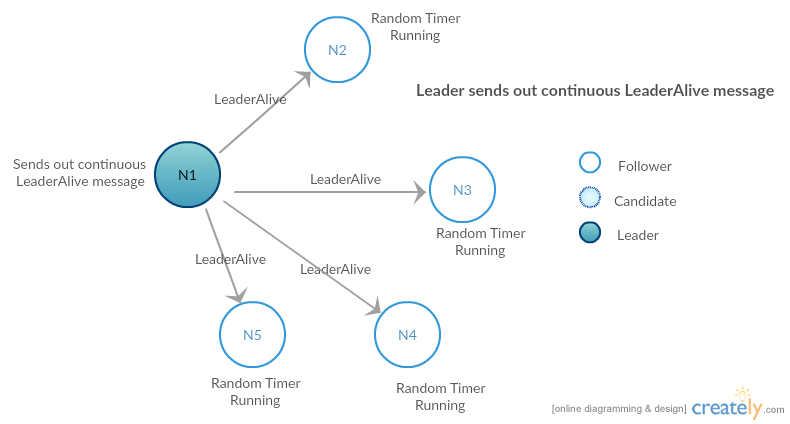
**Case 1:** Considering the case when there was timeout happened for only one of the nodes in the cluster. This is called as election term 1. Election term can be relate to as a global counter which keeps track of how many times the election has already happened and is useful in the situations where conflict arises.



The vote request message format is:

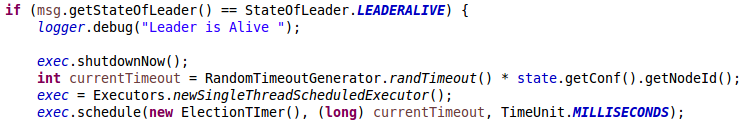


As soon as Candidate receives the majority of the votes (i.e. more than half the nodes current node is connected to) it declares itself as the leader of that cluster and sends out WorkMessage with StateOfLeader set to LEADERALIVE and Raft ElectionAction as LEADER, to all the nodes. Upon receiving this message other nodes updates their state with current leader ID and address to the new values. From now on the leader continuously sends out LEADERALIVE messages to all the nodes in order them to know that leader is alive.



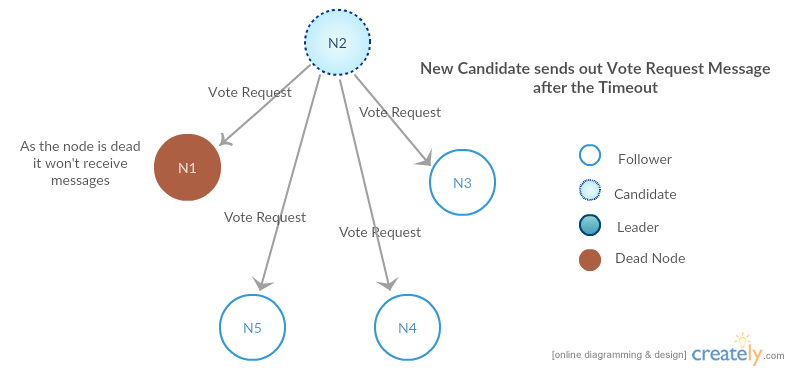
All the nodes executes a Randomized timer:

https://lh5.googleusercontent.com/10dwi4b6JCqp8Lw2x154-KfWzjKqX76Xrh4qyAH7sWOda8XD6E5ObSVkjRXagg9FkdpSXhC6SPmboOP2B1B77vkkVTaHD7PNr3_dY11C8yZekZdYplMbBe5kMm2nDfnwTBNcQ-zG



Which gets reset as soon they receive LEADERALIVE message in WorkMessage. The timer is important because it enforces that a new election gets triggered whenever the leader is dead.

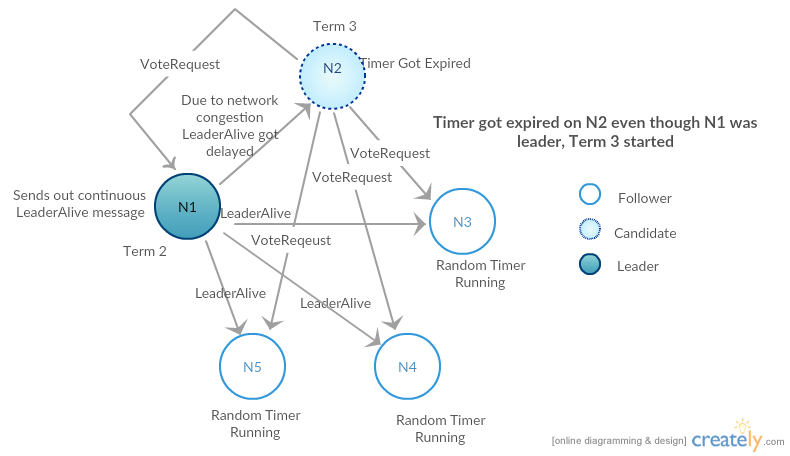
**Case 2:** Suppose the leader is dead and the timeout happened for one of the other nodes. This is Election term 2. The new node whose timeout just happened will change the state to Candidate state and broadcast the VoteRequest request to all other nodes. N2 might send out the vote request message to the dead node based on the fact that channel is still not removed from the node N2’s memory, but node N1 won’t receive anything. This process is called as Leader Re-election. The process is similar to the previous one and a leader will be elected based on the majority votes.



**Case 3:** Suppose N2 becomes the leader and N1 comes back to life. Now, as N1 was leader in earlier term. This will trigger re-election and the final leader will be elected based on whoever receives majority of votes.

**Case 4:** During an election if two nodes become candidate and sends out election vote request, the scenario will be handled using the Election Timeout Timestamp which is part of the RaftMessage. This scenario is already mentioned above.

**Case 5:** If the timer on one of the nodes gets expired because node did not receive LEADERALIVE message from the leader on time. The node becomes candidate and sends out vote request even though a leader already exists. The leader will check the term number with the new election vote request message and decide whether or not to vote for this request but as the term number will be higher the leader will vote and will become follower.



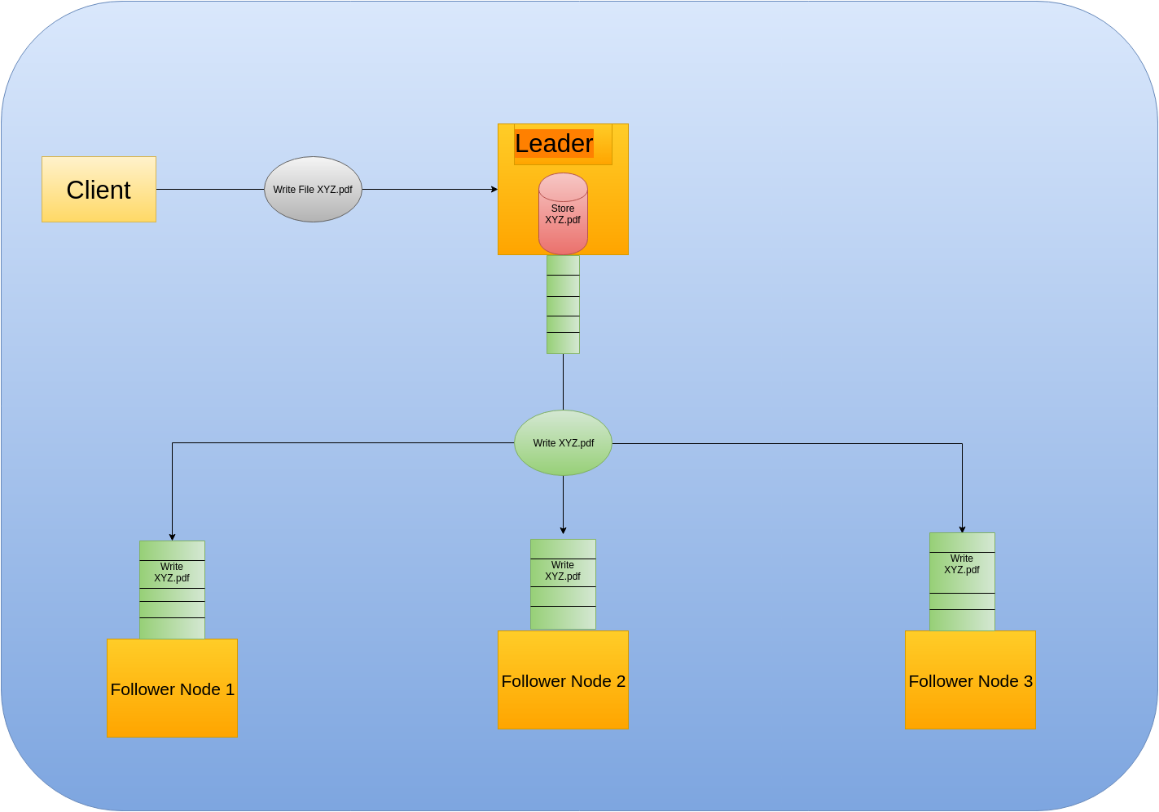
# REPLICATION

In a distributed file system, files can be stored on different servers and this introduces a disadvantage in the sense that if one of the nodes in the system crashes, it is possible that the files that were stored on the system could be lost, if they were not duplicated on any other servers. Replication is a good way of adding more fault tolerance in the system. We can replicate the files stored on a single server to multiple servers. In this case, shutting down of a single server need not mean that the required files have been lost from the system. In our approach, we have used data replication to duplicate the files stored on the leader node to all the other nodes in the system. This approach not only makes the system more resilient to crashes but also increases the no. of concurrent READ requests that can be serviced by the system by N-1, where N is the total no. of nodes in the system.

## 6.1 Data Replication Implementation:

The leader accepts WRITE requests from the clients and stores the requested files in his system. After it has completed persisting the file to his system, it sends the file to be replicated to all the nodes in the system.

To explain this concept in further detail, we assume that the client is connected the Leader and sends the chunked files to the Command Port of the Leader. The Leader node first stores the file in his instance of MongoDB. It then invokes the DataReplicationManager and calls the replicate () method, which runs asynchronously. The DataReplicationManager takes the file chunk as an input and creates a Work Message, which is relayed to the work port of the remaining nodes in the network. The remaining nodes in the network assimilate the incoming message by pushing it into the inbound work queue. The InboundWorkCommander at each node then evaluates whether it is a LEADER\_WRITE request. If so, it persists the file data contained in the received Work Message into its local MongoDB instance. As a result, all the nodes in the system now have the data which is stored at the leader.



## 6.2 Why Replication?

Replication increases availability of files in a file system. In the replication implemented in this project, each file is written on all the nodes, including the leader. This allows each node to service READ requests independently in the event that the other nodes crash or fail, as it has all the files required present in its system.

Another advantage is that since each file is replicated on every other node, the system can process multiple requests for the same file, for different clients concurrently. This would not be the case in a system where a file was stored on a single node, as clients would have to wait for the other clients, who have requested the same file, to complete their READ operation.

## 6.3. RIAK

RIAK is a distributed NoSQL database that stores key-value pairs. RIAK distributes data across multiple servers to ensure high availability. RIAK automatically distributes data around the cluster.

In the project, when we get a file that is less than 1 MB in size which means that the file has not been chunked, we store it into RIAK Database and that is replicated across other nodes.

In case the leader gets a Read request from the client which is less than 1 MB in size, the leader does not delegate the request to other nodes and directly fetches the RIAK database and returns it back to the client.

### Why RIAK:

The reason for using RIAK is because of its fault tolerance. It offer high read/write throughput and zero downtime. It is also very good when it comes to scaling out.

Riak.png

## 6.4. Mongo Data Base

In our project we must able to handle file of small size to large size. This needs persistence system to be scalable. NoSQL databases are used whenever we don’t know the size of data. Hence we have chosen Mongo for persisting data. Mongo is a document-oriented, cross-platform NoSQL database able to handle large data sets efficiently. It’s a scalable, high performance and highly available database. Everything in Mongo DB is stored as documents. Following are advantages of Mongo DB.

### Advantages of Mongo DB:

* Schema Less
* Deep query ability
* No complex joins
* Easy to Scale
* Auto-Sharding

## 6.5. Auto Replication

In our project when a new node is added into the system using node discovery, it initially does not have any of the data present in the remaining nodes. To make it consistent with the system, auto replication is used to replicate data present in each node into the new node. When a new node enters the system and discovers a node from the cluster, it sends a request to get its data. The recipient node processes the request and sends its data over to the new node.

# 7. Node Discovery

In distributed systems, there is a large number of machines that want to cooperate with one another to accomplish a common task. For example, sharing of work.  The first step in such an application will be to discover all available nodes in the network. This will be done using broadcast.

The new node that is trying to join the network first tries to find out all live hosts in the network. Upon discovery of live hosts, the new node broadcasts its Join Request message to all the live hosts. The nodes that are already a part of the network handle such request by completing 2 tasks:

* + The node that is already a part of the network adds the new node to its Edges and sends an acceptance message back to the new node.
  + The new node on receiving the message adds the already existing node to its Edges.

Apart from this process, auto replication also happens when a new node joins the cluster. One of the peer nodes starts transmitting data to new node asynchronously to update the persistence and cache.

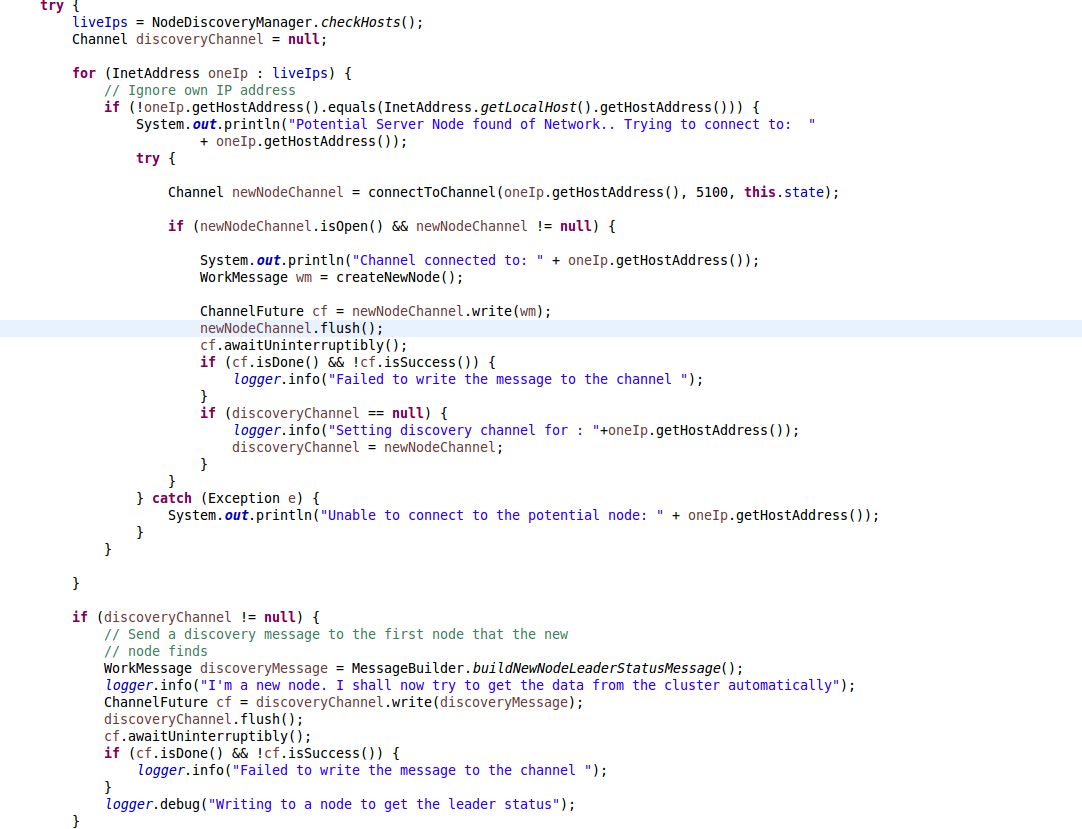
## 7.1 Why Node discovery:

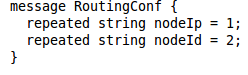
Node discovery is a very important part of distributed computing because using the node discovery, the already existing nodes in the network are able to connect to the newly connected node. This makes every node available in the network to know about every other node.

This also makes the entire system self-adapting in the sense that in case some nodes in the network are down and a new node is introduced, the system will automatically replicate data to it, inform other nodes about the new node and also the new node about all the other nodes, making it completely functional again.

NodeDiscovery.png

### Process

* A new node that joins the network first finds out all the live nodes in the network using the class NodeDiscoveryManager.
* After getting the list of potential nodes on the network, the new node tries to connect to each one of them.  
    
    
  

The already present node handles the request from the new node,adds the new node to its edges, creates a routing message which contains details about all its outbound edges and sends it back to the new node. The new node consumes this request, adds the discovered node to its edges. The information received from the discovered code can now be used to update the routing file of the new node, so that when the new node somehow gets dropped and joins back, it will already have information about the rest of the network.  
  
The protobuf **repeated** field type is being used to send the routing configuration.  
  
 ****



By using node discovery, we are able to detect when a new node has become a part of the network and then we replicate the data to the new node using auto-replication to ensure consistent data across all nodes.

# 8. Work Stealing

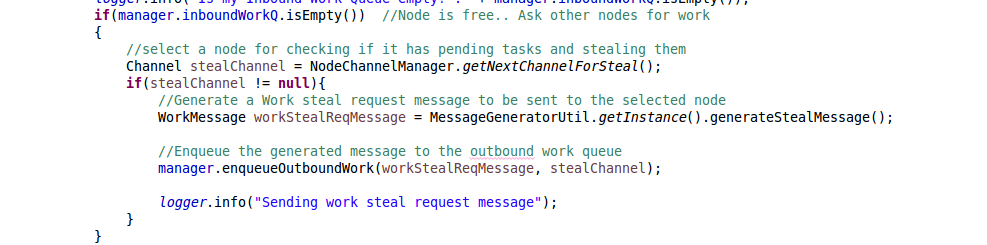
**WorkStealing.png**

Work stealing is the most interesting load-balancing methods for distributed systems because of its inherently distributed and scalable nature. It distributes the work over idle nodes and as long as work to do, no scheduling overhead occurs.

Each node on the network has a queue of work items to perform which is stored in its work queue.Whenever the inbound work queue is empty for a node, that node sends a Work Steal requestto other nodes and peaks into their work queue to find out if there is a pending Read task. If a Read task is found that task is dequeued and send back to the node that sent the steal request.

**Process:**

Each node after processing a task from its Inbound work queue, checks if its work queue is empty. If it is empty, the node chooses a node using the round robin ordering and sends out a steal work message to that node.



The node on receiving the steal request peeks its Inbound work queue to find if it has a pending read task. If there is a read task that task is dequeued from its queue and is sent to the requesting node.



# 9. Queuing

The Fluffy system consists of 6 different queues for handling different operations, at each node.

## 9.1. Why Queuing?

Every node in the network receives multiple messages on its command and work ports. Messages on the command port i.e. Command Messages are used by the client to communicate with the leader in the node.

Messages on the work port i.e. Work Messages are used for inter-node communication. They are of the following types:

* Heartbeat messages, sent between the leader and the other nodes to assess node status.
* Election messages, exchanged between the nodes to decide who the leader of the current term is.
* Replication WRITE requests and responses (which contain the file chunks to be replicated on the slave nodes).
* READ requests and responses which are sent between the leader and the slave nodes, to service READ requests from the clients.

Writing these messages on the node channels can lead to inconsistency and unavailability of the channels, as a huge number of messages could be asynchronously attempted to be written on them at the same time. Take for example, a client sends a READ request and the leader propagates this READ request to his slave. At the same time, another client has written a new file to the system, and the leader propagates this WRITE request from itself to all the slave nodes, including the one that got the READ request earlier. Note that this WRITE is asynchronous with the READ that occurred earlier. This means that it is possible that both these messages can be simultaneously written to the channel. Which message should be written to the channel first? How does the channel handle multiple messages written to it at the same time? In order to overcome these difficulties, a queuing system has been introduced in the network.

### 9.2 Types of Queues

Each node has the following 6 queues:

* Inbound Command Queue (ICQ)
* Outbound Command Queue(OCQ)
* Inbound Work Queue(IWQ)
* Outbound Work Queue(OWQ)
* Inbound Global Command Queue(IGCQ)
* Outbound Global Command Queue(OGCQ)

### 9.3 Advantages of Queuing

Each queue is synchronous blocking queue ([java.util.concurrent.LinkedBlockingDeque<E>](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingDeque.html)) that essentially queues messages synchronously. Multiple incoming messages are handled in a one-at-a-time approach, and all incoming messages are written to the queue at one time or another.

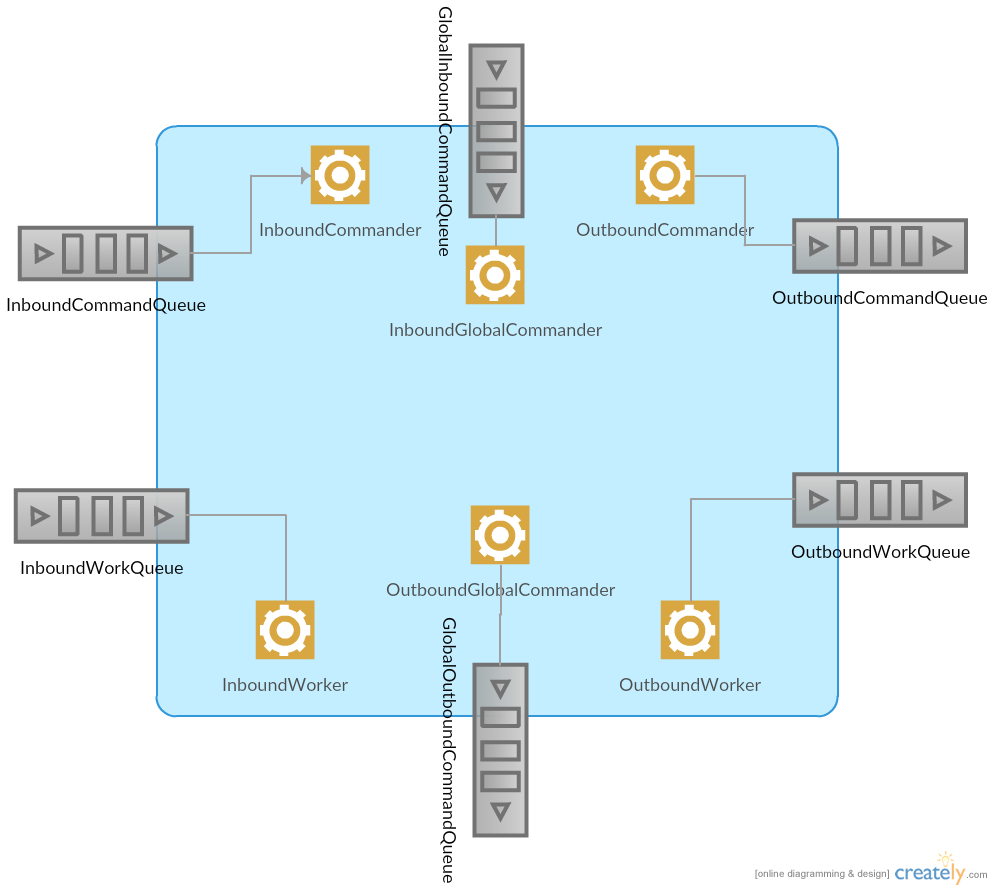
The take() operation on this queue is blocking, that is, the queue can block the currently executing thread for as long as there is nothing to take from this queue. This makes it an ideal data structure to approach our problem of simultaneous channel writes. The put () operation on this queue is also blocking in case we need to wait for space to become available in the queue.

Also, since this is a double ended queue, we can dequeue a message, try to write it to the corresponding channel, and if the channel write fails, put it back at the end of the queue, so that when we dequeue, we get the same message again. This queue is used as a layer over the channel to perform READS/WRITES simultaneously.

### 9.4 Additional Components

In order to process the contents of the queues, we have worker threads which continuously poll the queues for content and operate on it. There are 4 types of workers in our system, corresponding to each queue. Below are the types of workers and the corresponding queues they are associated with:

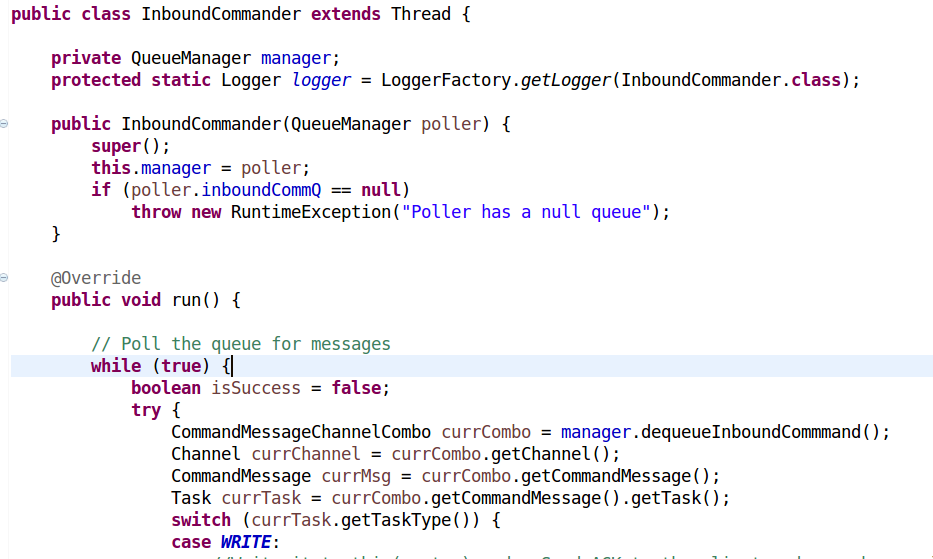
|  |  |
| --- | --- |
| **Worker Name** | **Associated Queue** |
| InboundCommander | InboundCommandQ |
| OutboundCommander | OutboundCommandQ |
| InboundWorker | InboundWorkerQ |
| OutboundWorker | OutboundWorkerQ |
| InboundGlobalCommander | InboundGlobalCommandQ |
| OutboundGlobalCommander | OutboundGlobalCommandQ |

Below is a diagram representing the queue system in each node.  


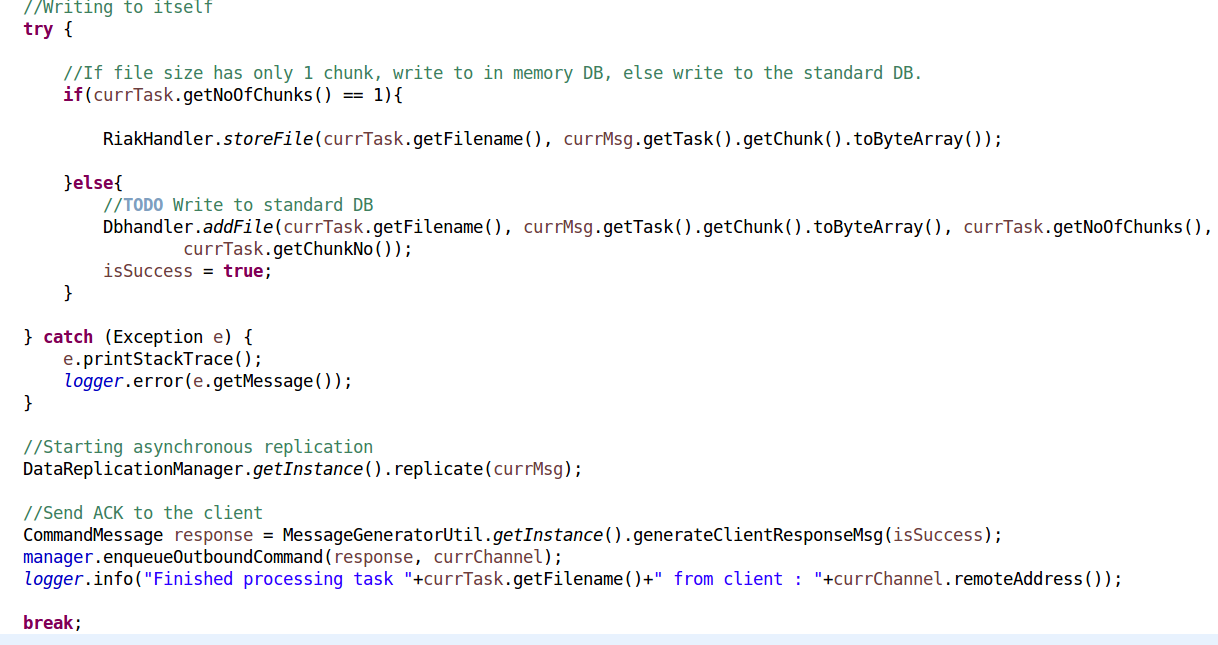
### 9.5 Detailed Explanation of the Queuing System

The functions of each of the queues and the workers are listed below:

1. **Handling Inbound Commands:** Inbound Command messages are sent by the client to the leader to perform READ/WRITE requests for a file(s). The CommandHandler accepts all incoming Command Messages, but blindly queues them into the ICQ. The following classes are used to handle them.
2. **InboundCommandQueue (ICQ):** When a client sends a request over the command port it is first received by the CommandHandler. The CommandHandler blindly enqueued this request into the ICQ (along with the client’s channel). This queue is instantiated by the QueueManager, who is started at server start up. It is consumed by the enqueueInboundCommand () and dequeueInboundCommand () methods, which are defined in QueueManager. The dequeueInboundCommand () calls the take () operation which is blocking, within a while () loop, so that it continuously blocks on the queue and processes it immediately when there is an element enqueued. This dequeue method is called by the InboundCommander(IC).
3. **InboundCommander(IC):** This is a worker thread that is started when the server is started initially. It continuously polls the ICQ and operates on the requests that are enqueued in it. It usually accepts tasks as READ or WRITE Command messages (from clients), manipulates them into WORK messages and then sends it to the OWQ. It also stores the client’s channel so that when the request is processed, it can retrieve it back and write on this channel. If a read request is received and the file is small enough to be read from the leader itself, then the IC generates a command message and writes the requested file from its RIAK instance into the output command message, which it sends to the OCQ, with the client channel to be written into.

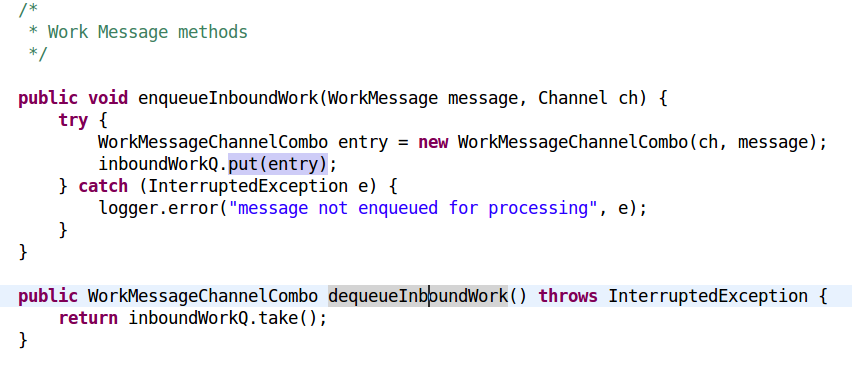


For WRITE requests, the IC first determines the no. of chunks in the input file chunk by looking at the noOfChunks field. If the file consists of a single chunk, the IC writes the file to its RIAK instance, otherwise it writes the file to its Mongo instance. The DataReplicationManager, which is running asynchronously then sends the file chunk received for replication to other nodes.



For READ requests, the IC first determines whether the file is present in the system. If it isn’t, it sends a request to the GlobalCommandAdapter, which communicates with all the other leader nodes in the network (inter cluster nodes), and asks for the file.   
If the file is present on the system, and has just 1 chunk, it retrieves it from its RIAK instance, encapsulates the file in a CommandMessage and sends it to the client.  
If the file has multiple chunks, it retrieves the channel which can service a READ request and prepares a work message, which is enqueued in its OutboundWorkQueue (OWQ). In this case, the client channel is stored by the leader, so it can be retrieved once the request has been processed.

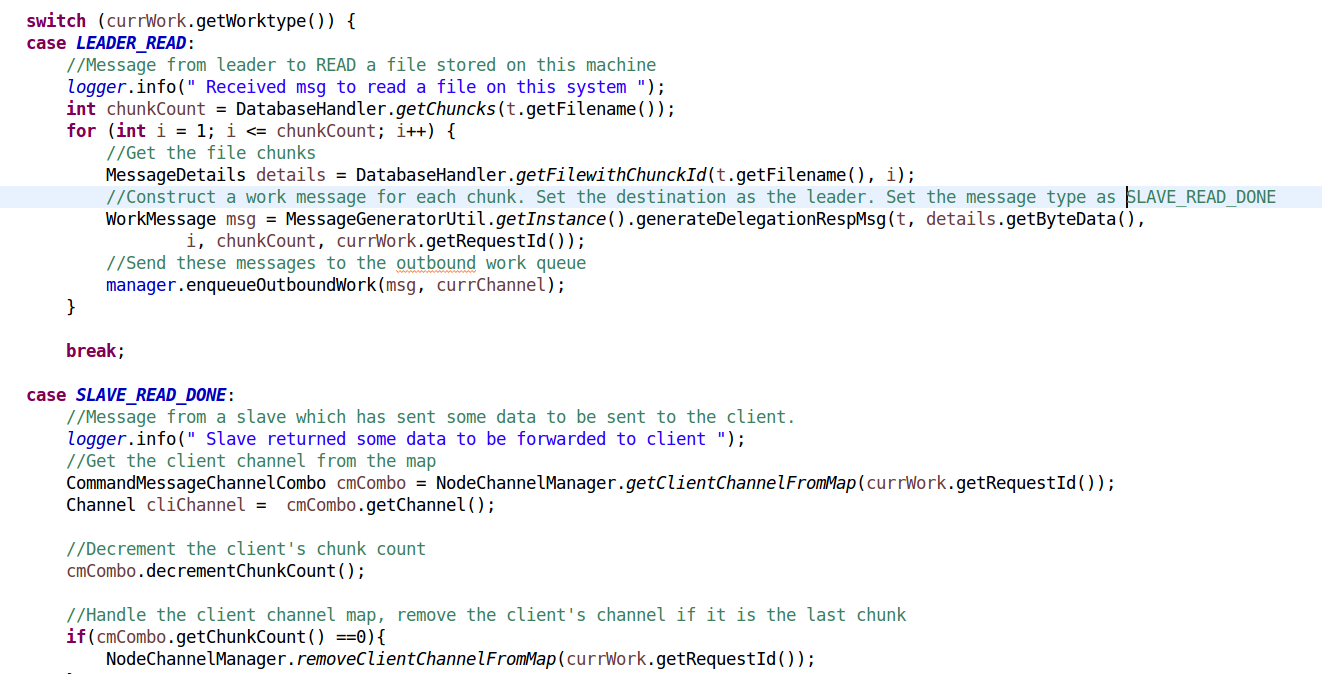
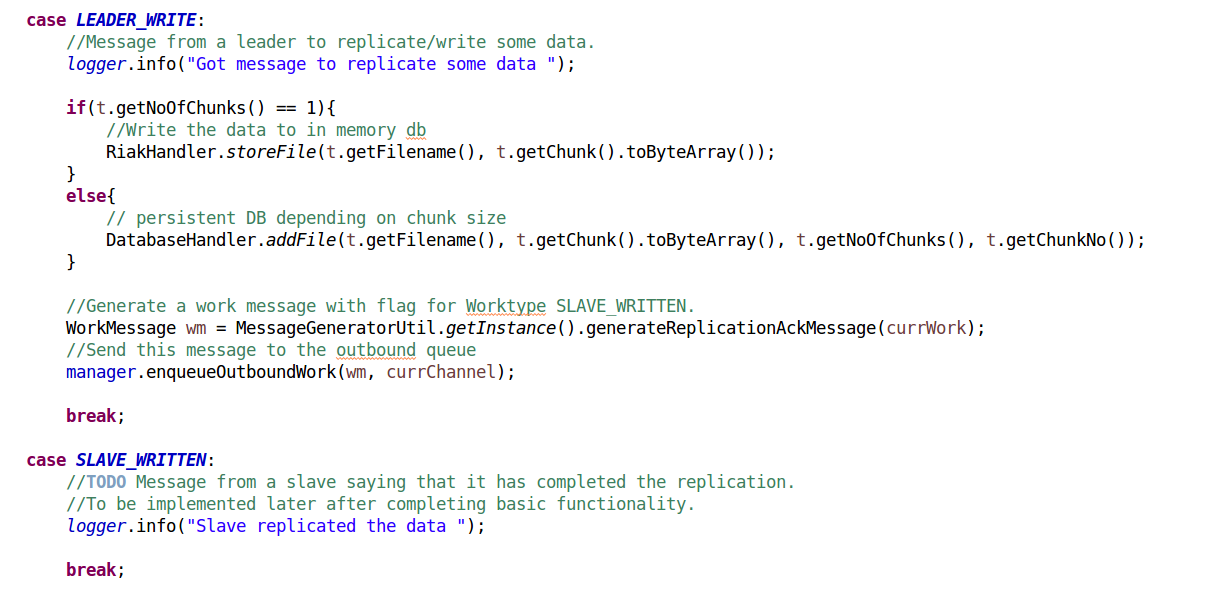
1. **Handling Outbound Commands:** Outbound Command messages are sent by the leader to the client. **The** following classes are used to handle them.
   1. **OutboundCommandQueue (OCQ):** This queue is used to send response messages to the client. It is enqueued when the leader has completed a WRITE operation successfully and wants to send a success response message to the client. It is also used when the leader has received a message from a slave that has serviced the READ response of a client. In this case, it prepares a CommandMessage to be sent to the client.
   2. **OutboundCommander (OC):** This is a worker thread that is started when the server is started initially. It continuously polls the OCQ and operates on the requests that are enqueued in it. It receives the message that needs to be sent to the client, along with the client’s channel. It then writes and flushes the message to this channel without doing any further processing on the message it has received. Its primary function is to prevent the client channel from being throttled with too many responses.
2. **Handling Inbound Work:** Inbound Work messages are sent between nodes in the cluster (intra-cluster messaging). The following classes are related with the inbound Work Messages in the system.
   1. **WorkHandler**: The WorkHandler initially accepts the Work Messages and decides what to do with them. If the WorkMessage comprises of a task, i.e. it involves a client-related operation, (and not a maintenance operation, such as leader election or node discovery) the WorkHandler enqueued it in the InboundWorkQueue (IWQ).   
      Other than this operation, the WorkHandler handles messages related to work stealing, leader election, ping messages and node discovery.
   2. **InboundWorkQueue (IWQ):** The IWQ contains READ/WRITE sent to the slave nodes from the leader to either READ a file or to replicate (WRITE) a file from the slave.



* 1. **InboundWorker (IW):** This is a worker thread that is started when the server is started initially. It continuously polls the IWQ and operates on the requests that are enqueued in it.  
     Firstly, it evaluates if the message received is a steal response message, i.e. the calling node has stolen a READ request from another node’s work queue. If so it processes it appropriately, and sends the requested file chunks directly to the leader.

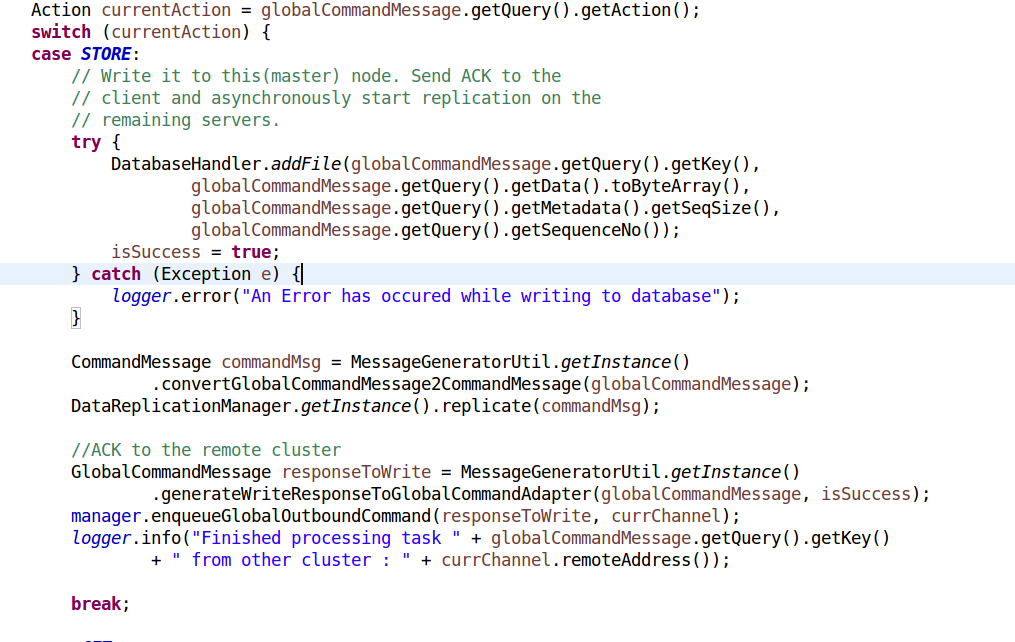
If the message wasn’t a work stealing related message, the node checks what kind of work is in it. It has the following 4 types:

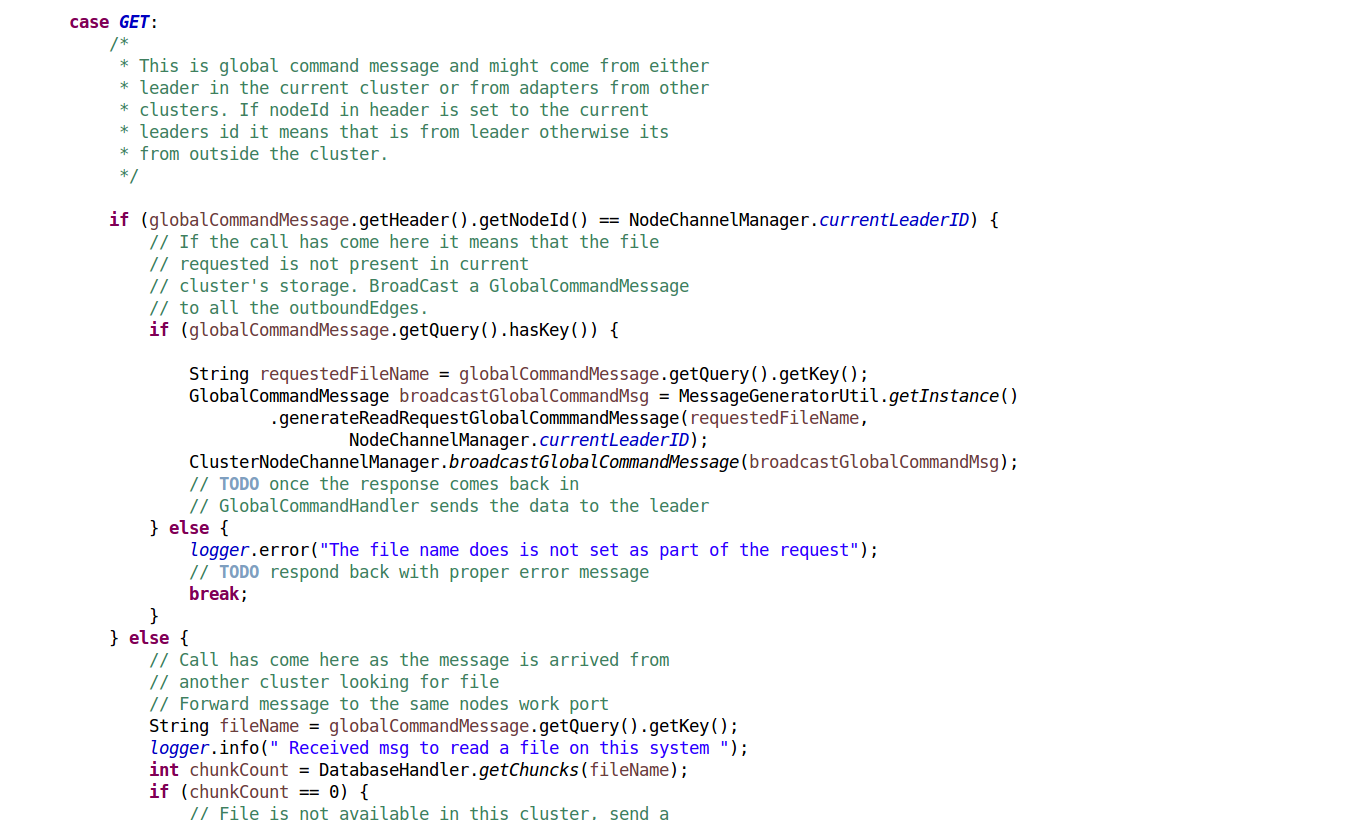
* + 1. **LEADER\_READ**: It’s a message to read a file stored on this node. In this case, the node gets the relevant data chunks from its Mongo instance and sends it to the OWQ.
    2. **SLAVE\_READ\_DONE**: This is a message from a slave to a leader, and the slave has serviced a READ request. Since the node servicing this message is a leader, it retrieves the client’s channel and generates Command Messages that are sent to it OCQ, along with the client’s channel.
    3. **LEADER\_WRITE**: This is a message from a leader to a slave to replicate some data held in the message. Depending on the noOfChunks field, the IW then stores it to RIAK or to Mongo.
    4. **SLAVE\_WRITTEN**: This is a response from a slave to a leader telling it that it has successfully replicated the data asked to be replicated.

After the IW has processed this request, it checks its own IWQ size and if it is empty, it generates a steal request to another node to process a READ request on that node.

1. **Handling Outbound Work:** Outbound Work messages are sent between nodes respond to READ/WRITE requests for a file(s). The following classes are used to handle them.
   1. **OutboundWorkQueue (OWQ):** This queue is used to send response messages to the leader from slave nodes for servicing READ or replication requests. It is also enqueued from the DataReplicationManager in the leader to send work messages to the slave nodes. It is also used when the GlobalCommandAdapter notifies the leader that it has found the file that was requested earlier by the leader. The leader also uses this queue to send a delegation request to a slave node to service a READ request. It is widely used by the IW, to notify the sender that processing of the current message is completed. In the case of work stealing, an overloaded node also uses this queue to signal a stealer node that it can take a task from its IWQ.
   2. **OutboundWorker (OW):** This is a worker thread that is started when the server is started initially. It continuously polls the OWQ and operates on the requests that are enqueued in it. It receives the message that needs to be sent to the recipient, along with the recipient’s channel. It then writes and flushes the message to this channel without doing any further processing on the message it has received. Its primary function is to prevent the recipient channel from being throttled with too many responses.
2. **Handling Global Inbound Commands:** GlobalInbound Command messages are sent from the leader to communicate with the node hosting the Global communication adapter. The following classes are used to handle them.
   1. **InboundGlobalCommandQueue (IGCQ):** The GlobalCommandHandler enqueues all requests into this queue to be handled by the InboundGlobalCommander for sending requests to the node hosting the Global Adapter.
   2. **InboundGlobalCommander (IGC):** This class is used only on the node that hosts the Global Adapter. It processes the messages coming on the global port and retrieves the current action of the message. It can be of 2 types :
      1. STORE:  In this case the message is from a node from another cluster who wants to store a file on this cluster. The file is written to the current node and is replicated on the other nodes in the cluster.

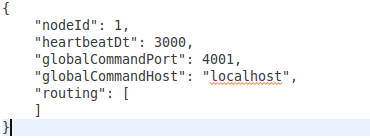


* + 1. GET: This is used in the following cases. A node from another cluster requests a file from this cluster. The leader has experienced a GET request but the cluster does not have this file. Therefore, it must request the node having the Global Adapter to search for this file among nodes in other clusters. If the sender is the leader itself, the Global Adapter contacts other clusters to retrieve the file. If not, the node retrieves the file from its storage and sends it to this channel.  
       

1. **Handling Global Outbound Commands:** Global Outbound Command messages are used to send responses to the Global Adapter that the cluster has completed processing the messages that require inter cluster communication. The following classes are used to handle them.
   1. **OutboundGlobalCommandQueue (OGCQ):** Messages are enqueued into this queue from the IGC or the IC to notify the requesting cluster that it has written the file, or to send the requested file to the outer cluster. It is used by the OGC to write the requests directly on the channel that is allocated along with the message.
   2. **OutboundGlobalCommander (OGC):** This class is used only on the node that hosts the Global Adapter. It continuously polls the OGCQ and operates on the requests that are enqueued in it. It receives the message that needs to be sent to the client, along with the remote cluster’s channel. It then writes and flushes the message to this channel without doing any further processing on the message it has received. Its primary function is to prevent the remote cluster channel from being throttled with too many responses.

# 10. Inter-Cluster Communication

The global commander is a concept which we introduced to incorporate inter-cluster communication. A Global Command Adapter runs on all the nodes in the cluster but leader gets connected to only one configured in the **GlobalRoutingConfig.conf.**

****

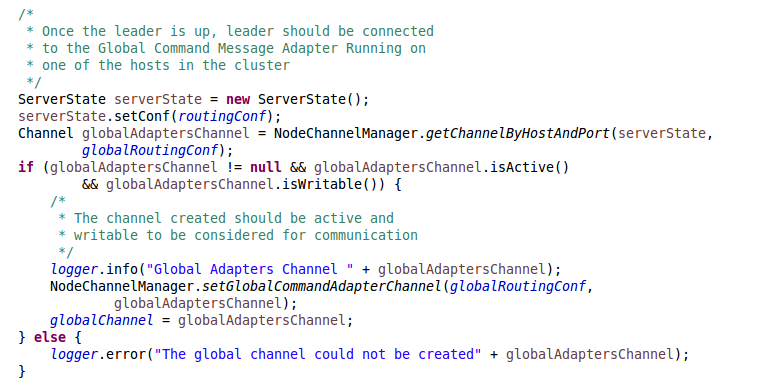
The adapter runs on the global command port and the global command host defines which nodes adapter is current adapter we would like to expose to outside clusters. The routing holds all the other nodes global adapters host/port pair.



The **MessageServer** class initiates separate global worker and boss threads using **Server Bootstrap** class.

10.1 Working of Global Command Adapter

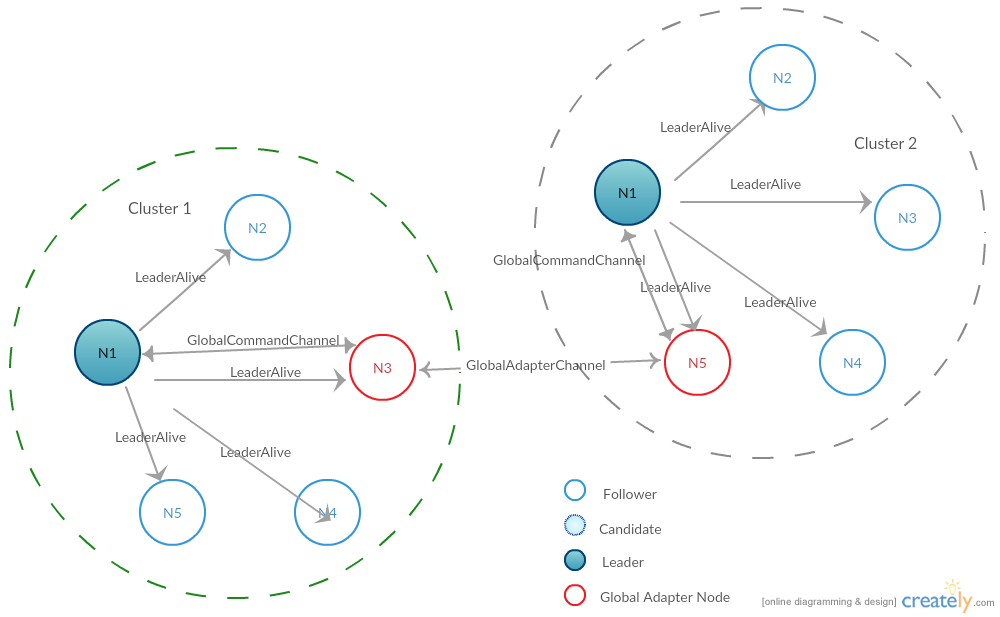
As soon as the leader election completes and a new leader gets elected. The first task it does is to create a channel connection to the adapter i.e. leader connects to the host and port given in the GlobalRoutingConfig.conf



**Diagram: Code to show global connection after leader get elected (Class:ElectionManagement.java)**

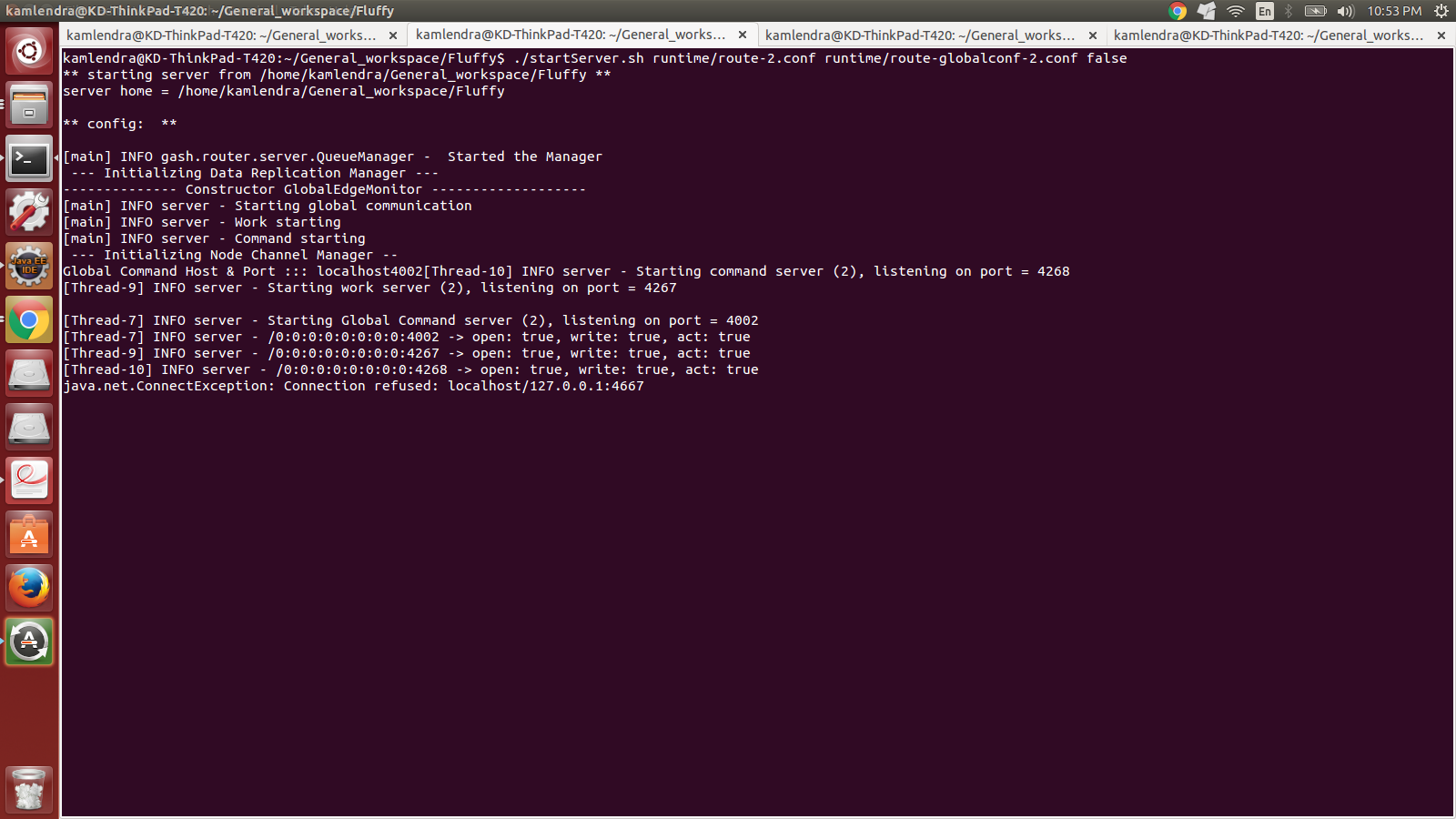
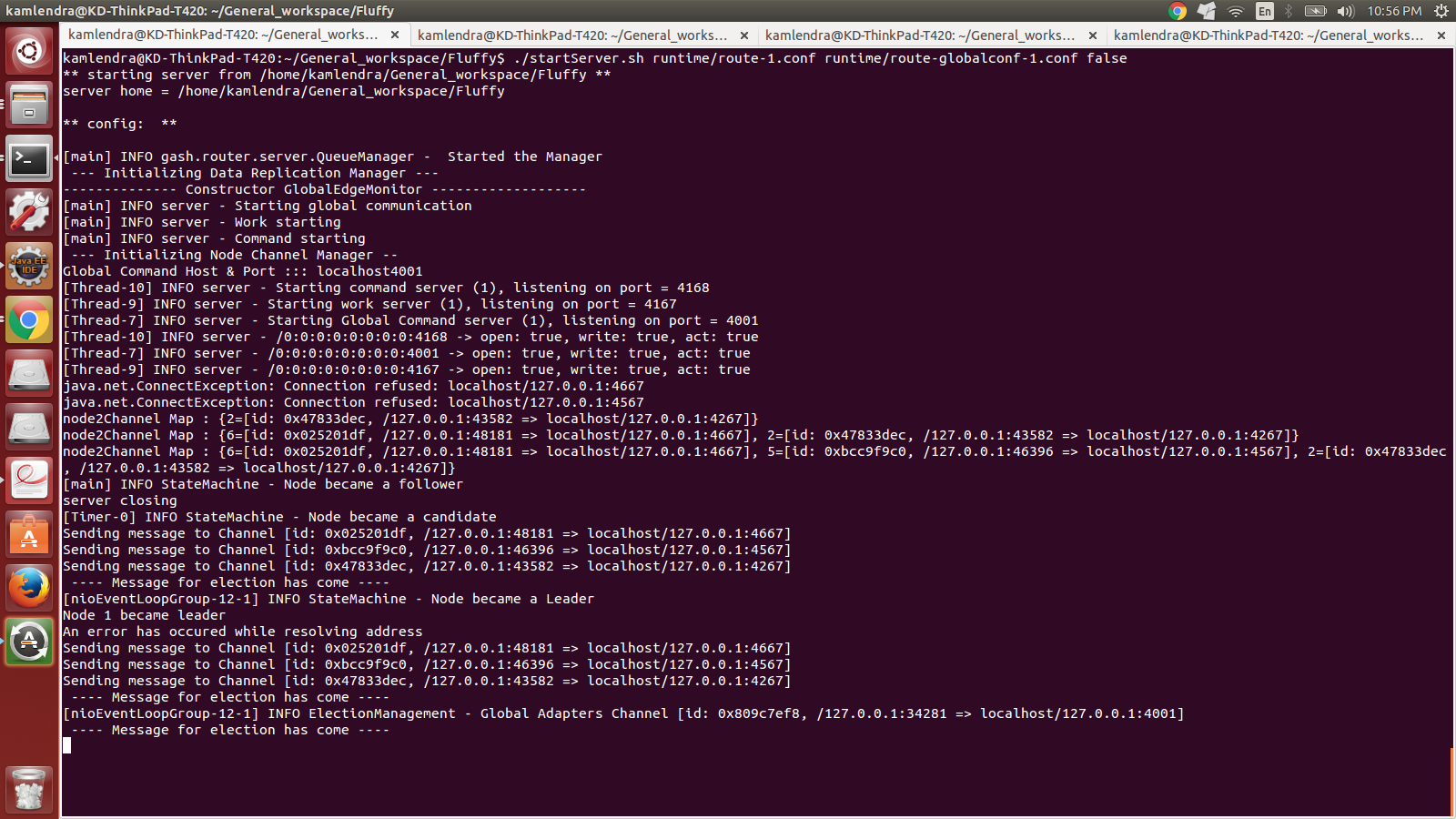
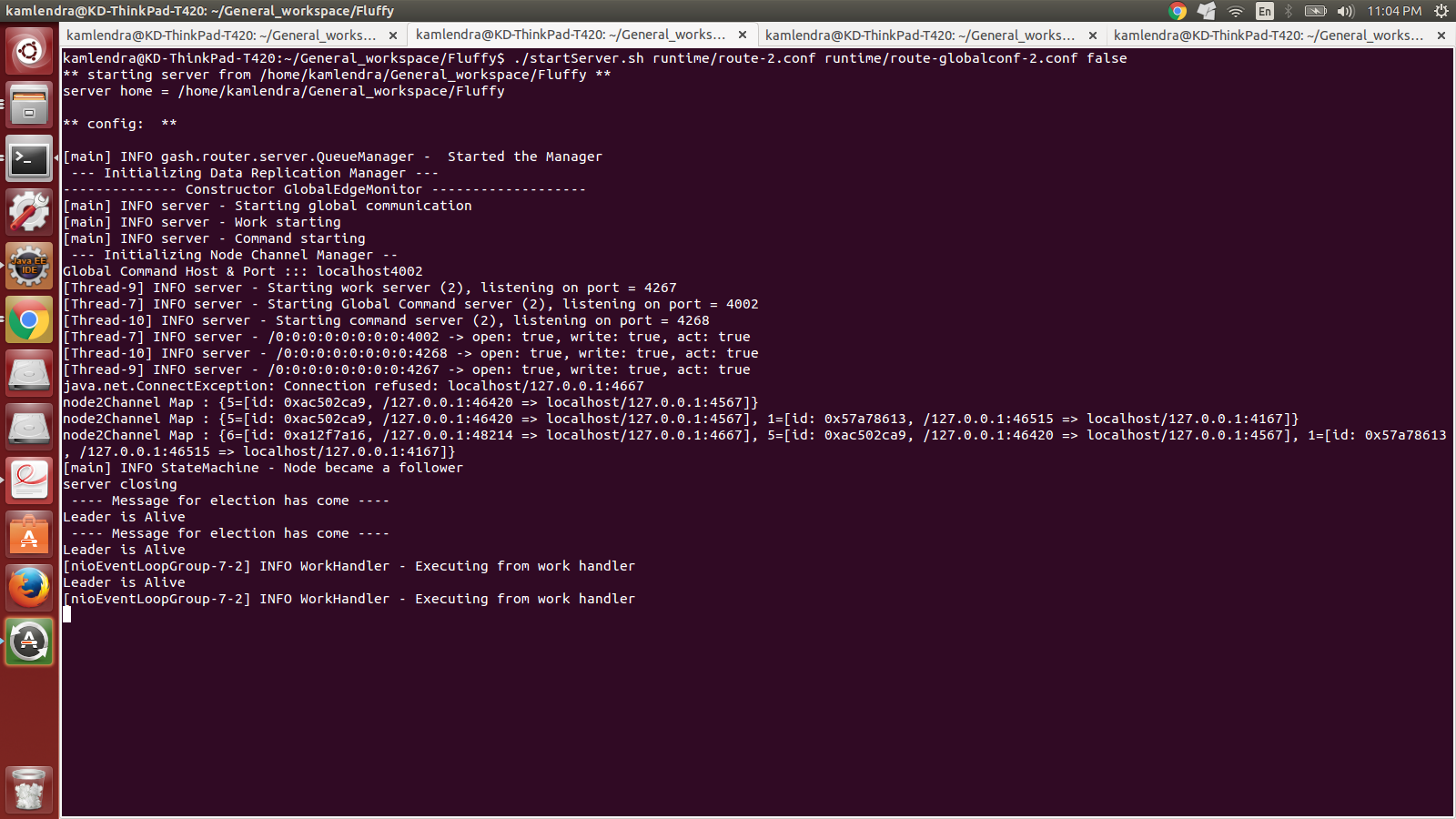
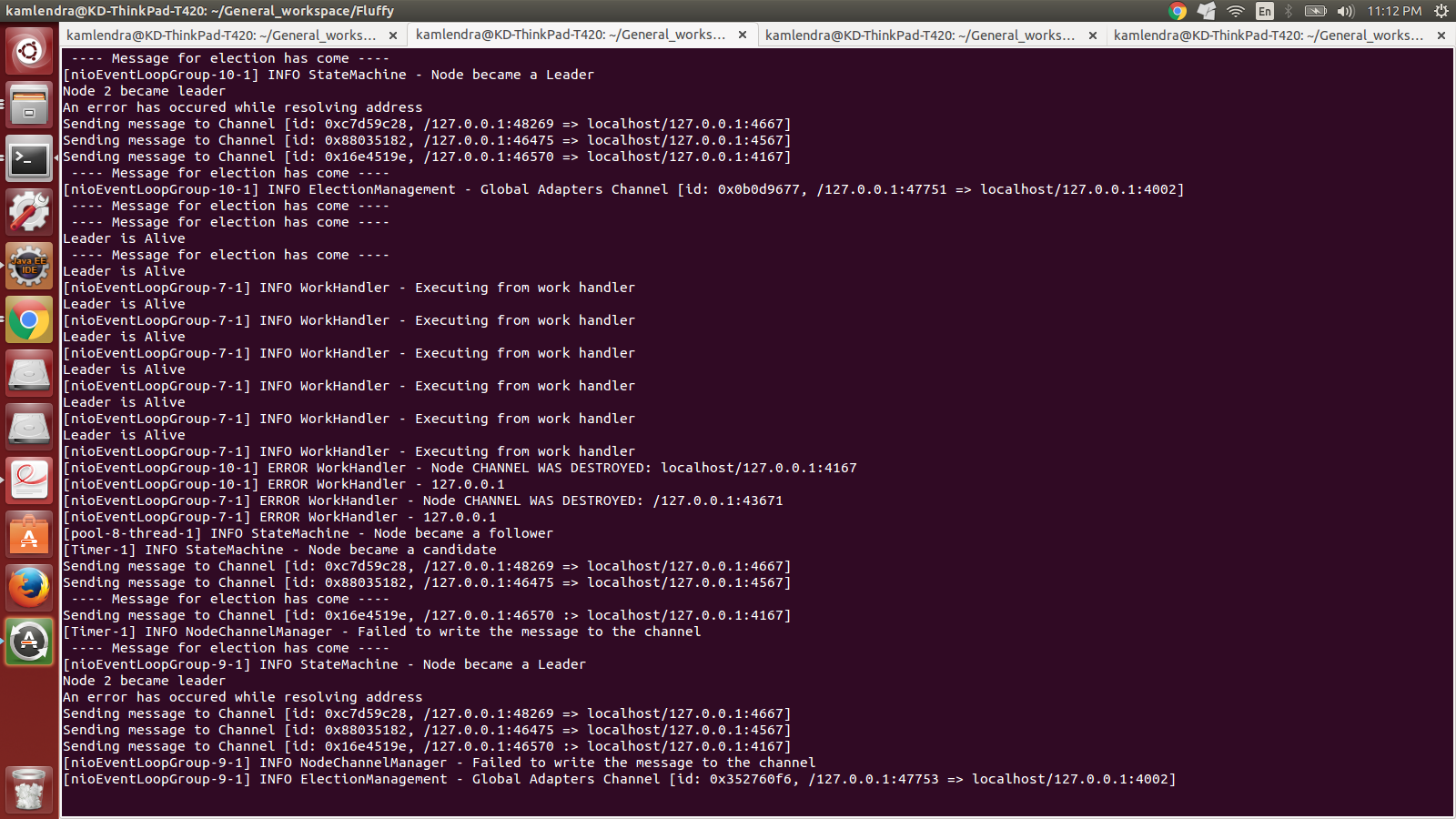
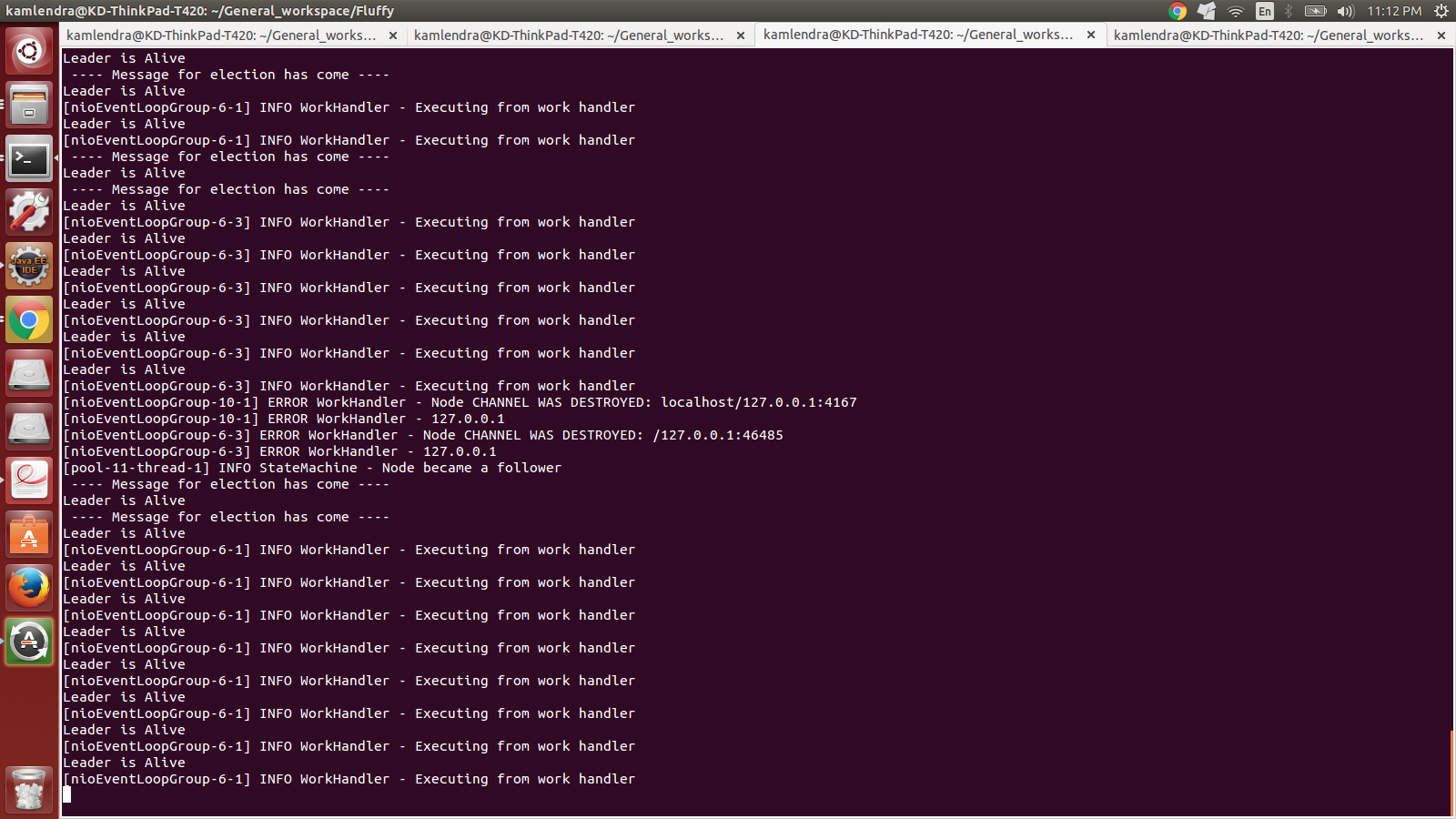
The global command adapter is useful in following scenarios:

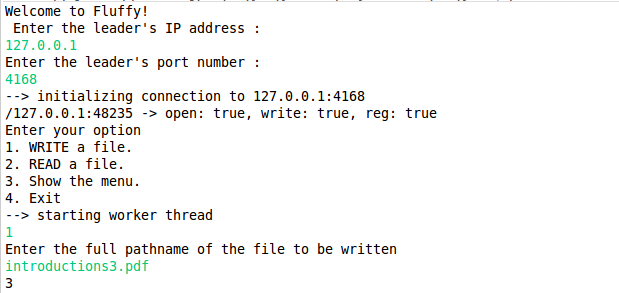
1. When leader receives a READ request and the requested file is not available in current cluster. The leader converts the incoming CommandMessage to GlobalCommandMessage and forwards it to the GlobalCommandAdapter. Upon receiving a message, adapter A1 broadcasts this message to all the clusters nodes connected to current cluster C1. The adapter A2 in other cluster C2 has that file. A2 responds back with GlobalCommandMessages based on the number of chunks for that particular file. As soon as A1 receives the response it converts the message to a WorkMessage and forwards it to the leader in order to respond back to the client who requested the file.
2. When a GlobalCommandMessage arrives from A2 to A1 requesting READ for a file. The adapter A1 checks its own database for the file as the adapter is running on one of the nodes in the cluster and knows about all the files. If the file is available A1 will starting sending all the chunks to A2.



# 11. Client API

In our project we have implemented Java and Python Client API. Both clients provide option to upload and download file. Upon receiving the file client chunks the date in 1 MB chunks and prepares CommandMessage for each chunk. As we have discussed previously about workflow of client request. Here we have inserted screen shots of working of client (python and java) as well as server.

1. Server starting  
   
2. Node 1 became the leader.  
   
3. All other nodes are getting the heartbeat  
   
4. Node 1 got dropped and node 2 became leader  
   
5. Node 2 became leader now

6. Client 

# 12. Test Cases

**Server side functionality**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Test case** | **Steps** | **Expected result** | **Test case result** | **Status** |
| 1 | Leader election | 1. Start all the servers.  2. Check on each server how the leader election progresses. | Currently elected leader ID is displayed on all the terminals. | Currently elected leader ID is displayed on all the terminals. | Pass |
| 2 | Write large File | 1. Start all the servers.  2. Send a large file(> 1MB and <20MB) from the client to be written. | File is written to leader’s Mongo instance and slave’s Mongo instances. | File is written to leader’s Mongo instance and slave’s Mongo instances. | Pass |
| 3 | Write small File | 1. Start all the servers.  2. Send a large file(<= 1MB) from the client to be written. | File is written to leader’s Riak instance and slave’s Riak instances. | File is written to leader’s Riak instance and slave’s Riak instances. | Pass |
| 4 | Read existing file from system. | 1. Start all the servers.  2. From the client, ask for a file that exists on the system. | Leader gets file from a slave node and sends it to the client | Leader gets file from a slave node and sends it to the client | Pass |
| 5 | Read non-existent file from the system. | 1. Start all the servers.  2. From the client, ask for a file that does not exist on the system. | Leader sends a request to the GlobalAdapter to get the file. | Leader sends a request to the GlobalAdapter to get the file. | Pass |
| 6 | Node discovery | 1. Start all the servers.  2. Start a new node that has no entries in its routingEntries section. | A node in the system recognizes this node and adds it to the cluster. The new node gets the remaining nodes in its nodeChannelMap as displayed on the console. | A node in the system recognizes this node and adds it to the cluster. The new node gets the remaining nodes in its nodeChannelMap as displayed on the console. | Pass |
| 7 | Auto replication | 1. Start all the servers.  2. Start a new node that has no entries in its routingEntries section. | The new node gets information about the new nodes, and its Mongo & Riak instances gets populated with data on the other nodes’ Mongo & Riak instances | The new node gets information about the new nodes, and its Mongo & Riak instances gets populated with data on the other nodes’ Mongo & Riak instances | Pass |
| 8 | Work Stealing | 1. Start all the servers with 1 leader and 2 followers.  2. Simulate a situation in which 1 follower node is overloaded with READ requests and the other node has processed at least 1 READ/WRITE request. | The idle node peeks into the busy node’s queue and retrieves the READ request and services it and sends the result back to the leader. | The idle node peeks into the busy node’s queue and retrieves the READ request and services it and sends the result back to the leader. | Pass |

**Java Client side functionality**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Test case** | **Steps** | **Expected result** | **Test case result** | **Status** |
| 1 | Upload file | 1. Start the Java client.  2. Enter the IP address and port of the leader.  3. Select the option to upload the file.  4. Enter the fully qualified pathname of the file.  5. Press ENTER. | Leader writes the file to its Mongo/Riak instance and file is written to the follower nodes via replication. | Leader writes the file to its Mongo/Riak instance and file is written to the follower nodes via replication. | Pass |
| 2 | Download file | 1. Start the Java client.  2. Enter the IP address and port of the leader.  3. Select the option to download the file.  4. Enter the fully qualified pathname of the file.  5. Press ENTER. | File is read from a follower node and is written to the client inside the client directory. | File is read from a follower node and is written to the client inside the client directory. | Pass |

**Python Client side functionality**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Test case** | **Steps** | **Expected result** | **Test case result** | **Status** |
| 1 | Upload file | 1. Start the Python client.  2. Enter the IP address and port of the leader.  3. Select the option to upload the file.  4. Enter the fully qualified pathname of the file.  5. Press ENTER. | Leader writes the file to its Mongo/Riak instance and file is written to the follower nodes via replication. | Leader writes the file to its Mongo/Riak instance and file is written to the follower nodes via replication. | Pass |
| 2 | Download file | 1. Start the Python client.  2. Enter the IP address and port of the leader.  3. Select the option to download the file.  4. Enter the fully qualified pathname of the file.  5. Press ENTER. | File is read from a follower node and is written to the client inside the client directory. | File is read from a follower node and is written to the client inside the client directory. | Pass |

# Learnings

* Netty
  + Understanding the Netty working was most challenging part as it was new to all of us.
  + We spend about 1 week to completely understand Netty framework
* Protobuf
  + It was very interesting to learn the structure of message in protobuf.
    - Connection and Replication in RIAK
      * Connecting and storing files to RIAK was new to all of us in the team.
      * We learnt how to insert and retrieve files from RIAK.
      * We also learnt how useful it is to use RIAK.
    - Leader Election
      * We have implemented RAFT. It was very challenging to implement RAFT.
      * We face lot of difficulties in implementing RAFT.
      * We were able to overcome difficulties and implement RAFT
* Connecting with peers
  + We were able to connect to other cluster using Global Adapters which used a common proto file
* Architecture
  + The major learning includes how to design and implement an architecture and take decisions when facing a challenge.

# Future Work

1. The Outbound queues are currently processing the enqueued messages synchronously, even though most of the time, the outbound messages need to be written to different channels. The outbound queues could be modified in the sense that the QueueManager could hold a list of outbound queues in a dictionary/map structure and retrieve it from there, to be written. This could increase throughput for file read/writes as well as client communication.
2. As of now replication is propagating to all the nodes on the network. Although this provides resiliency, it is inefficient as data is redundantly being stored on all the nodes. We intend to replicate data on a few of the nodes using some processing logic.
3. The work port of every node is overloaded with different types of messages (node discovery, leader election, replication and read requests and responses). We intend to create a separate port for at least the replication and read messages, so that they don’t conflict during channel read/writes with the other types of messages.
4. Introduction of log replication and consensus in the RAFT architecture.
5. When the client sends a WRITE request but the space available in current cluster is not sufficient then the request should be forwarded to another cluster. Leader converts the CommandMessage to GlobalCommandMessage and forwards it to its own adapter A1. A1 forwards it to one of the adapters. The adapter checks the same and does the intended operation.
6. Introduction of redirection logic so that the client is redirected to the leader node in the network, so that it does not have to know the leader beforehand.
7. Introduction of direct READ from the follower node so that the leader need not handle the delegation response, and the data is passed directly from the follower node to the client, instead of through the leader.
8. Introduction of redirection from the client to the node that can service its READ request.

# Conclusion

We able to successfully complete “Fluffy” a distributed file storage system. Due to asynchronous APIs of Netty we were able to handle huge amount of request. Protobuf allowed us to easily communicate between cross-platforms. We were able to store and retrieve files of varying size using Mongo DB and RIAK. To conclude we integrated various technologies build an efficient and scalable file storage system.

# References

[1] <http://netty.io/>

[2] <https://developers.google.com/protocol-buffers/>

[3] <http://maxondev.com/serialization-performance-comparison-c-net-formats-frameworks-xmldatacontractserializer-xmlserializer-binaryformatter-json-newtonsoft-servicestack-text/>

[4] <http://raft.github.io/>

# Contributions

* + - 1. Savio Fernandes - Architecture design, Queuing system, Replication, READ/WRITE processing, Auto-replication, Research.
      2. Kamlendra Chauhan - Leader election, Architecture design, Global communication adapter (inter cluster communication), Refactoring code, Message Generation, Research.
      3. Akshatha Anantharamu. - Python and Java client, Architecture design, Work stealing, Mongo adapter, Testing, Research.
      4. Himanshu Jain - Architecture design, Node discovery, Work stealing, Auto-replication, RIAK adapter, Testing, Research.
      5. Pramod K. - Report creation, testing, Architecture design, Research.