

CMPE 275

Gossamer (Project 1)

[[https://github.com/virajnilakh/DistributedFileStorageAndSharing]](https://github.com/virajnilakh/DistributedFileStorageAndSharing)

**Team Members:**

Ashutosh Singh

Viraj Nilakh

Shruti Padmanabhan

Table of Contents

[1. Introduction 4](#_Toc481515768)

[2. Architecture 4](#_Toc481515769)

[3. Technologies Used 5](#_Toc481515770)

[a. Netty 5](#_Toc481515771)

[b. Google Protocol Buffers 5](#_Toc481515772)

[c. Redis Server 5](#_Toc481515773)

[4. Topology 5](#_Toc481515774)

[a. Advantages of Mesh Topology 6](#_Toc481515775)

[b. Disadvantages of Mesh Topology 6](#_Toc481515776)

[5. Leader Election 6](#_Toc481515777)

[a. What is Raft? 6](#_Toc481515778)

[b. Why Raft? 6](#_Toc481515779)

[c. Scenarios: 6](#_Toc481515780)

[i. Cold Start: 6](#_Toc481515781)

[ii. All nodes at once: 6](#_Toc481515782)

[iii. Two nodes turn Candidates altogether: 7](#_Toc481515783)

[iv. Vote request to Leader: 7](#_Toc481515784)

[v. Leader heartbeats + follower heartbeats 7](#_Toc481515785)

[vi. Dynamic Node addition 7](#_Toc481515786)

[vii. Dynamic Node deletion 7](#_Toc481515787)

[viii. Our approach: 7](#_Toc481515788)

[ix. Difficulties faced: 7](#_Toc481515789)

[6. Workflow 8](#_Toc481515790)

[a. Node Discovery 8](#_Toc481515791)

[b. WRITE request from client directly connected to cluster 8](#_Toc481515792)

[c. READ request form client directly connected to cluster 8](#_Toc481515793)

[d. READ request from client outside cluster 8](#_Toc481515794)

[7. Replication 8](#_Toc481515795)

[8. Work Stealing 9](#_Toc481515796)

[9. Client API 9](#_Toc481515797)

[10. Learning 9](#_Toc481515798)

[11. Important Design Decisions 9](#_Toc481515799)

[12. References 10](#_Toc481515800)

[13. Contributions 10](#_Toc481515801)

[14. Special Work 10](#_Toc481515802)

[15. Future Improvements 12](#_Toc481515803)

[16. Project Build process 12](#_Toc481515804)

[17. Project Run process 13](#_Toc481515805)

# Introduction

Gossamer is our version of the distributed file system like Dropbox that we are currently familiar with. Similarly, our system is also capable of storing, retrieving and balancing tasks/ data to make file sharing/storing to be decentralized over a network of servers.

Clients can retrieve or send any file type (pdf, mp4, mp3, docx, jpeg, png) to the system without worrying about fault tolerance, scalability or work load. Since we process requests asynchronously, user experience would be more pleasant. We have tested files for upto 800MB written to cluster, retrieved it and the movie was playable to the end.

# Architecture

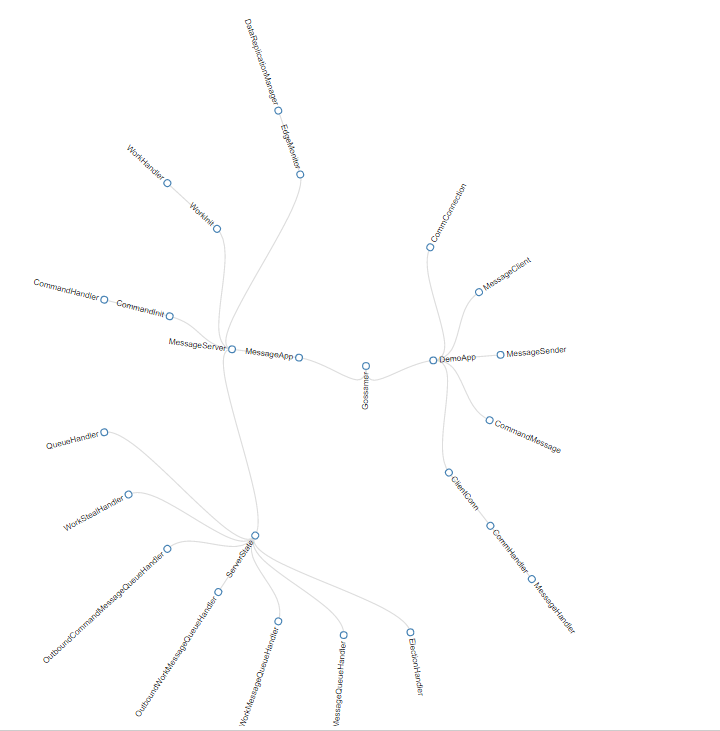


Diagram: Architecture tree showing main classes and behavior

Detailed code architecture description can be found at [[https://gladiatorash.github.io/gossamer]](https://gladiatorash.github.io/gossamer).

# Technologies Used

Following is the technology stack we chose:

|  |  |
| --- | --- |
| **Languages**: | Java |
| **Core Packages**: | Netty, Google Protobuff |
| **Storage/ DB**: | Redis, MySQL |
|  |  |

Let’s take a closer look at why choosing those packages and storage choices were in fact advantageous for our system:

# Netty

Netty is a “client-server framework which enables quick and easy development of network applications such as protocol servers and clients” [1]. It has a generic API for different transport types that supports, blocking and non-blocking of sockets. It gives total control over customizing threads. It’s event model provides room for separation of concerns making it more flexible and easy to extend.

Some of the other features would be:

* Easy to follow Java documentation with examples
* Performance wise, gives better throughput and lower latency
* Huge save on memory

# Google Protocol Buffers

Protobuf is a language and platform independent “mechanism for serializing structured data” [2]. It is comparatively, much “smaller, faster and simpler” than XML[2]. Once you define a structure in a dot proto file, on building, you generate a dot java file with methods to access those fields.

# Redis Server

Jedis is a Redis handler built in Java as a key-value pair but better as it can store richer set of data types (list, set, sorted set, hashes) “and different levels of on-disk persistence and provides high availability via Redis Sentinel and automatic partitioning with Redis cluster” [4].

Other advantages of Redis:

* Fast performance – about “110000 SETs per second, 81000 GETs per second” [3].
* Operations are atomic – concurrently two clients could access it
* In built feature support publisher-subscriber model
* In memory database

# Topology

We decided to implement Mesh Topology where in all the nodes would be connected to every other node. But it’s quite expensive because we are making multiple redundant connections.

# Advantages of Mesh Topology

Simultaneously you could transmit multiple messages and it can handle high traffic. Even if one connection fails, there are alternative routes to communicate the message over. Adding or deleting nodes from the cluster isn’t an issue at all and it could be done without disrupting the existing connection between nodes.

# Disadvantages of Mesh Topology

As mentioned earlier, there is high redundancy of network connections. Hence it’s an expensive topology to maintain as compared to others. Also, setting up and maintaining the network is extremely difficult.

# Leader Election

Leader Election is a process of assigning a single machine as a coordinator to distribute tasks among multiple nodes. There are certain conditions that a leader election algorithm must follow:

* **Termination**: the algorithm should run only for a finite duration to pick a leader.
* **Uniqueness**: there should not be more than one leader in a cluster.
* **Agreement**: all the nod es must be aware of the who the leader is.

We decided to implement Raft as our leader election algorithm.

# What is Raft?

Raft is a consensus algorithm. “The difference is that it's decomposed into relatively independent sub-problems, and it cleanly addresses all major pieces needed for practical systems. We hope Raft will make consensus available to a wider audience, and that this wider audience will be able to develop a variety of higher quality consensus-based systems than are available today” [6].

# Why Raft?

Since Raft assumes that each node is connected to all the other nodes, which is exactly what we have developed through Mesh topology, we could directly request votes, get vote responses, send heartbeats…etc. It’s highly scalable as it doesn’t need to know the number of nodes in the cluster.

**Cold Start:**

When there is only one node in the cluster, then it becomes the leader. When another node joins this cluster, it starts receiving heartbeat from the leader. When the leader goes down in this case there is a reelection and new node becomes the leader.

**All nodes start at once:**

When all the nodes start at once, they first get connected to each other and then election takes place. When another node joins this cluster, it starts receiving heartbeat from the leader. When the leader goes down in this there is a reelection and new node becomes the leader.

**Two nodes turn Candidates altogether:**

In this case the node whose timer times out causes the second node to become follower and cast the vote for him. If one node is at lower term then the other then it becomes the follower and casts vote.

**Vote request to Leader:**

When a node asks for a vote from the leader, the leader provides his response of him being alive and turns the candidate back to follower state. This happens even when the new node joins the cluster for the first time with and active leader. But if the leader is at a lower term then the candidate then it becomes the follower.

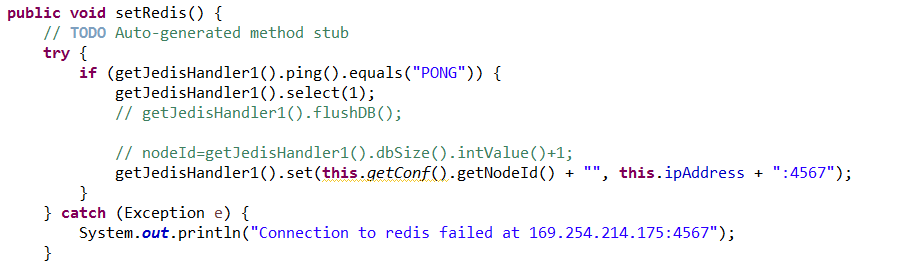
**Leader heartbeats + follower heartbeats**

Both Leader and followers send heartbeats to each other. Getting heartbeat from leader, causes follower’s timer to reset and thus preventing reelections. Getting heartbeats from followers, prevents them from being deleted from the cluster and causes resetting of their timer

**Dynamic Node addition**

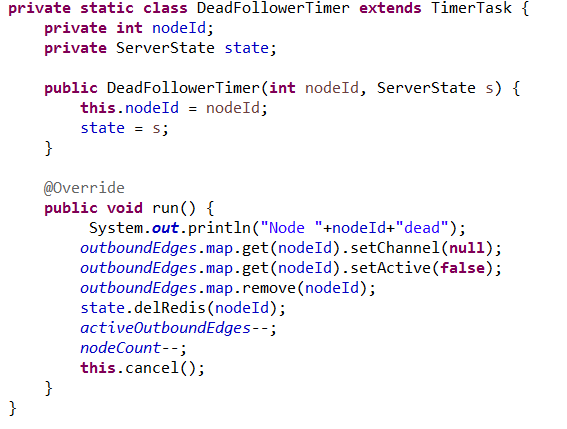
As soon as a node boots up, it is automatically added to the cluster and connected to all the nodes in that cluster. If a leader is present, it starts getting heartbeats from the leader. If the leader is not present an election takes place. Used replication on redis to prevent single point of failure.

state.setRedis();



**Dynamic Node deletion**

When a node stops receiving heartbeats from any other node for a particular timeout period, that node gets deleted from the cluster dynamically. Used replication on redis to prevent single point of failure.



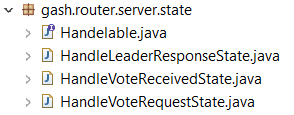
**Our approach for node addition and deletion:**

To use 3 replicated redis servers for node registration, as soon as a server boots up. When a new node is registered, all the nodes become aware of that and try to connect to the new node in a mesh topology. When a node is deleted from redis, all other nodes become aware of that and they delete its entry from their connection list.

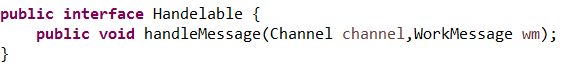
Our novel idea of using redis for node discovery was accepted by the class for getting the details of their next cluster Id in a ring topology

**State Design Pattern**

In order to get rid of the constant if else if conditions, we implemented state design pattern to handle election messages. Due to this each node would handle election messages differently based upon its state i.e Follower, Candidate, Leader.



All the server states implemented the common API Handelable.java to providing common functionality of handleMessage() depending upon its state



**Global Communication**

Our cluster can handle global ping and global write. It knows when to stop the ping or write request. It can also detect a change of leader from the next cluster dynamically.

**Difficulties faced:**

* Whole of the semester time, seemed to be wasted, trying to debug raft algorithm when the problem was with the machine’s firewall causing the raft implementation to fail constantly. After turning off the firewall, all of the raft implementation worked swiftly as expected.
* To make redis server accessible by other hosts from different machines, its configuration file needed to be changed.
* Sometimes the host was unable to connect to redis and the server used to crash constantly without us figuring out the reason why. So we created a PingRedis class which will test redis connections for all the servers beforehand, eliminating the of pain restarting the servers and assuming that redis connection will be successful the next time.
* When the protobuf files were changed so that every cluster has the same protobuf implementation, our entire working code failed. Problems like msg entering the if condition even when it didn’t satisfy the if condition came through. It took a lot of time to solve that problem.
* When added a new functionality the previous code used to break. That caused us to revert to our old code many times and to rewrite the code once again, carefully debugging every step, to find out what did go wrong.

# Workflow

# Node Discovery

Since it’s a distributed system, it should always look around its peripheral to capture systems that are constantly trying to connect to the cluster. As a first step, we are broadcasting message to all the nodes nearby for a 30 second window. After the discovering step, it would write to a route config file for the nodes to form mesh network.

After forming a network, leader election process starts. Now say a new node wants to join this network, it can be added only by a leader.

Note: This is redundant now due to usage of global redis, nodes on start will connect to global redis and update the statuses of their leaders

# WRITE request from client directly connected to cluster

Client sends a write request to cluster requesting leader to distribute the file across the network.

The first step involves client chunking the file and storing it in an in-memory Array List. This arrayList is then converted into Command Request messages which are then passed to an executor service which spawns as many threads as the OS allows and writes these messages to the channel to be sent to the leader.

# READ request form client directly connected to cluster

Client sends a read request to leader requesting a file by name. Leader replies with the file chunks and client sorts these chunks and assembles them to a file.

# READ request from client outside cluster

Client sends a requestAllFilesNames request to cluster to get a list of all files stored on the cluster.

# Replication

In distributed system, files are usually split into different nodes and there is always a fear of what happens when a node fails and we end up losing that data. Hence it’s very crucial that copies of the file are stored across multiple nodes.

We are using w=1 replication as it would be easier implementing work stealing as all nodes would have the data and any node can steal message request from a queue of any other node. We are using Redis nodes at each cluster node to keep track of all chunks.

# Work Stealing

We observed that, as we are already replicating files chunk by chunk across all the followers, each follower does equal job process write request. But only for the case of a read request it becomes an overhead only for the leader, as we are serving all the read requests to be performed by the leader. So inorder to relieve leader from most of the pain of read request, we use work stealing on read requests. Whenever a leader receives a work steal request, it will update its stealRequest variable to true, thus ignoring all the steal request further received until that stealRequest is served and reset to false. The followers are programmed to issue a steal request after every 7 seconds so that all the nodes would get equal chances to process the read request and only one cluster would not be overwhelmed by that. Used LinkedBlockedDeque for achieving this. Created separate threads for inbound and outbound communication.

Created separate threads for inbound and outbound work and command messages. Create queue handler for addressing all queue related operations. Created WorkStealHandler thread to create and send steal requests.









# Client API

Client allows four options to the user.

Option 0: Exit

Option 1: Write file to cluster

Option 2: Read a specific file from server

Option 3: Fetch File Names stored in cluster

Option 4: Ping to cluster

Option 5: Read Multiple files from cluster

# Learning

* Netty
* Redis
* ProtoBuff
* Architecture advantages and disadvantages
* Leader election algorithms
* Java multithreading handling
* Working in a team

# **Important Design Decisions**

1. Executor Service configured to use maximum threads system can support. This makes sending faster as max threads are created to dispose queue work.
2. Client uses a LinkedBlockingDeque capable of being read from both ends
3. Recreating files at both server and followers to confirm that file is transferred without errors. To be removed after testing.
4. Lamport clock maintained at each node to keep track of all requests inserted into db. Will be used to synchronize in future, if any aberrations.
5. Replication is done to all followers as limited time did not allow us to build sharding which could complicate searching files
6. Work stealing implemented as all followers steal from leader's queue and if unable to service the message due not being fully synchronized or any other reason, returns the message back to leader of the cluster
7. Work stealing is made easier by doing it only on reads and not writes. Previous design decision of replicating to all nodes helps in making work stealing from any node. Currently implemented to only steal from leader node.
8. When a client requests a file, leader checks if it is present in the cluster. If it isn't, passes the request to the next cluster if it exists or throws an exception if it doesn't.
9. All clusters have been provisioned an ID. Our cluster ID is 2. Client retains the ability to connect to a cluster of its choice.
10. All clusters use a common redis db to store their leader node addresses. Client can connect to any cluster via this common redis db.
11. Config file to start server only stores nodeID which can be abstracted away in the future.
12. Although code for cold start was written, it is not being used because of implementation of global redis which removed it's need.
13. Redis db at server being used to store metadata of all chunks and files. Planned to store hot data but limited in memory storage making it difficult. MongoDB could be used to store the hot data but it's usage is restricted by project scope.
14. All files are saved with their metadata, filenames are MD5 hashed and stored. If file with same filename is sent, it gets overwritten as in a normal file system. This implements updating of files.
15. Follower can directly reply with file chunks to the client thus preventing overloading of leader. Uses work stealing to steal read request from leader and sends chunk to client directly.

# Special Work

**Parallelization:**

Our team experimented with different classes available in Java to make file sending parallelizable.

**1st Approach:**

Using a single thread to wait on queue and transfer chunks.

**2nd Approach:**

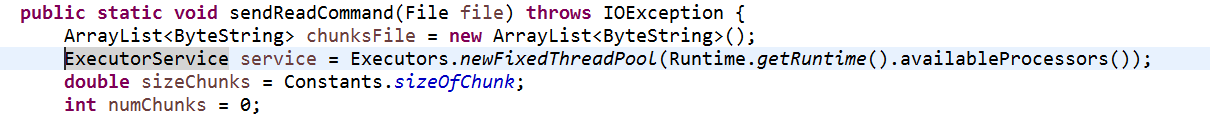
Using Java Concurrency API’s ExecutorService to create a single thread and send chunks

**3rd Approach:**

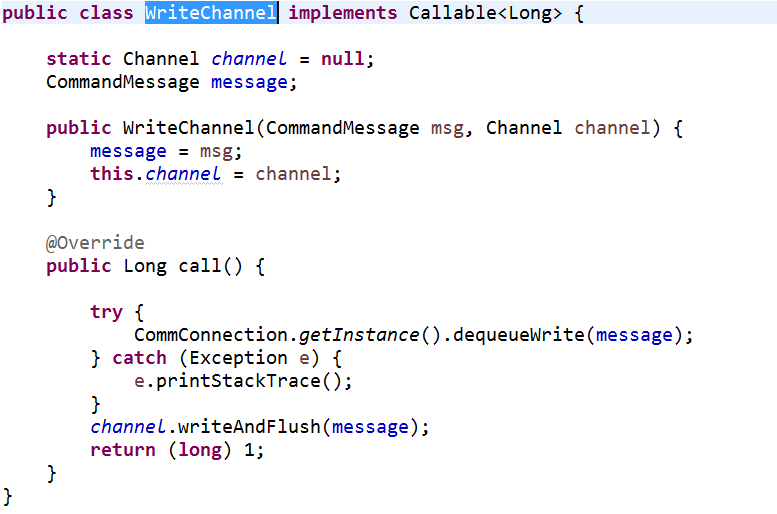
Using Executor Service with Callable and Futures to keep track of acks.

**4th Approach:**

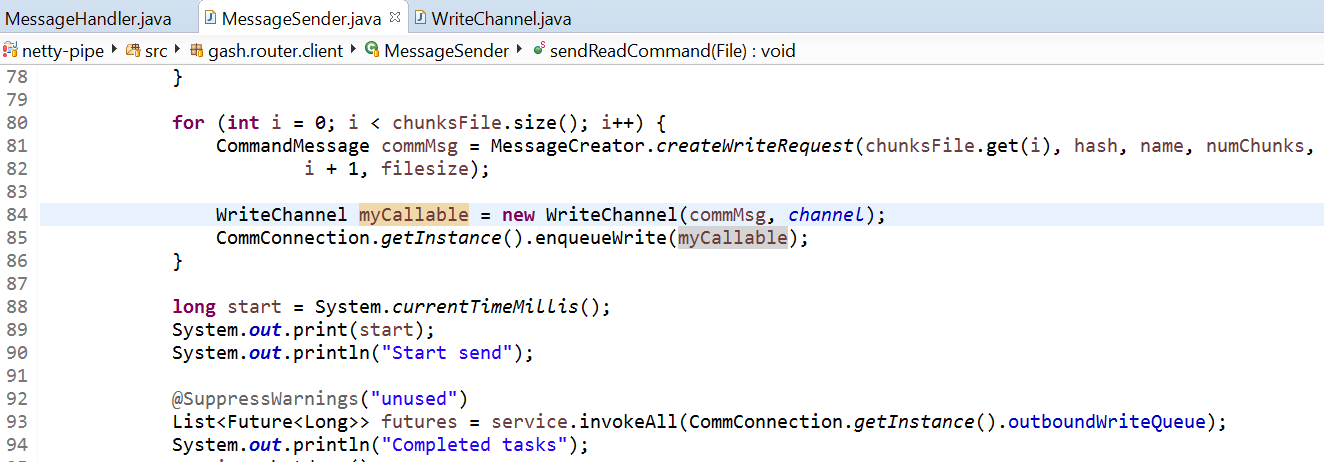
Using Executor Service with max threads as



along with callable and futures



And invoking these max threads to send chunks from client to server.



This made sending large files a small task but an obvious drawback is the queue which cannot store randomly large files. But it was a design decision to make and we did it this way. Thus sending files beyond which cannot be accommodated in the heap set when server is started causes OutOfMemoryException.

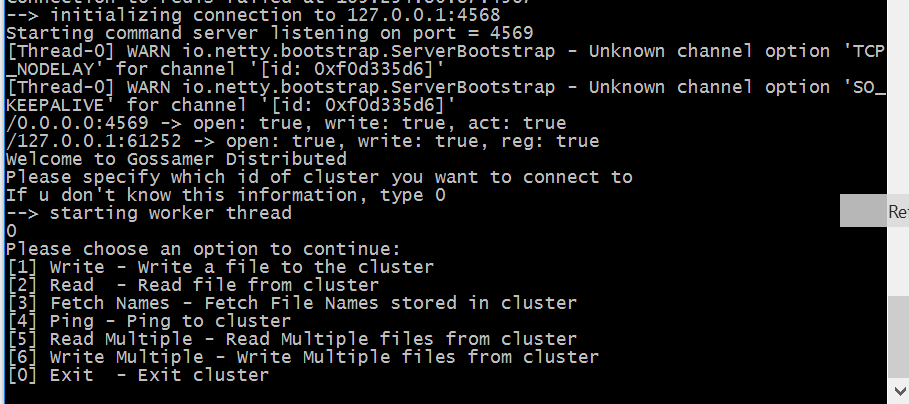
This can be prevented by increasing memory from 1000 MB **as set below,** to larger as can be accommodated by the OS and available memory space.

**java -server -Xms1000m -Xmx1000m -cp ".;lib/\*;classes/" gash.router.server.MessageApp runtime/route-7.conf**

# Future Improvements

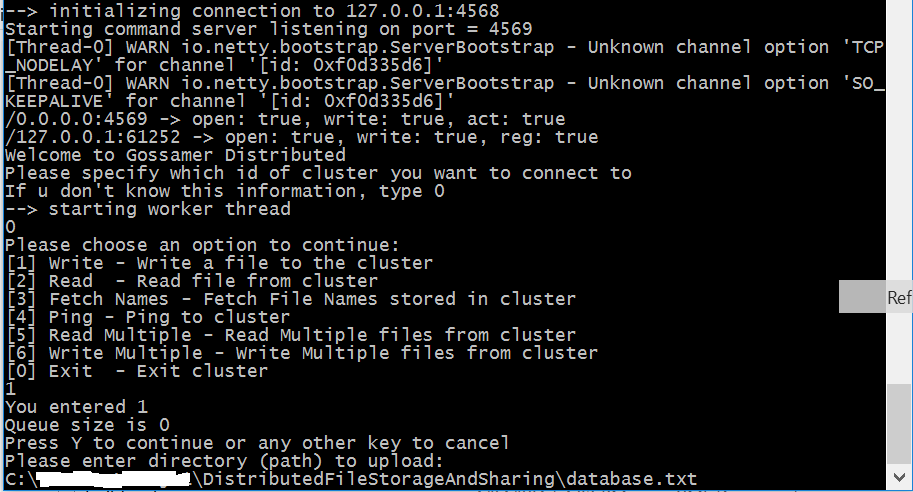
1. Making log replication to followers from leader. Currently raft is used for leader election. Heartbeat message can be modified to send logs as well.
2. Sharding to store partial file in each of the nodes. Currently all files are being stored on all nodes, thus providing resiliency. Also, instead of using W=1, using W=N/2+1 where W= no. of nodes file chunk is written to and N=No. of Nodes.
3. Using state pattern as on the server side specially to make the code cleaner. Currently, we have divided the classes functionally.

# Project Build process

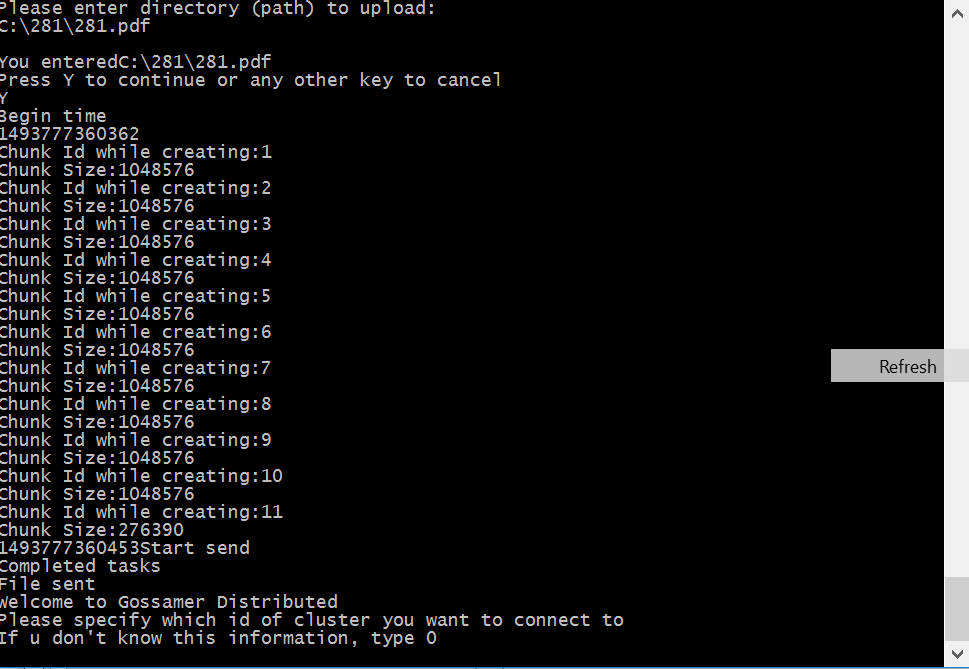
1. Start Redis server present at / Redis-x64-3.2.100/ redis-server.exe from cmd by running the following command
   1. redis-server.exe redis.windows.conf
   2. Use Flushall to clear data whenever restarting server
2. Start MySQL server and run /database.txt commands to create db. Update password and username of MySQL in /src/database/DBHandler.java on Line 24
3. Set your follower and leader IPs in /src/global/Constants.java on Line 10
4. Build Gossamer project.
5. In /runtime/route-7.conf update your nodeId. Note that routing section can be kept empty as it is unused.
6. Run server by java -server -Xms500m -Xmx1000m -cp ".;lib/\*;classes/" gash.router.server.MessageApp runtime/route-7.conf
7. Run client by java -client -Djava.net.preferIPv4Stack=true -cp ".;lib/\*;classes/" gash.router.app.DemoApp localhost 4568
8. 

Client should have options on screen. Choose options and operate the system.

# Project Run process

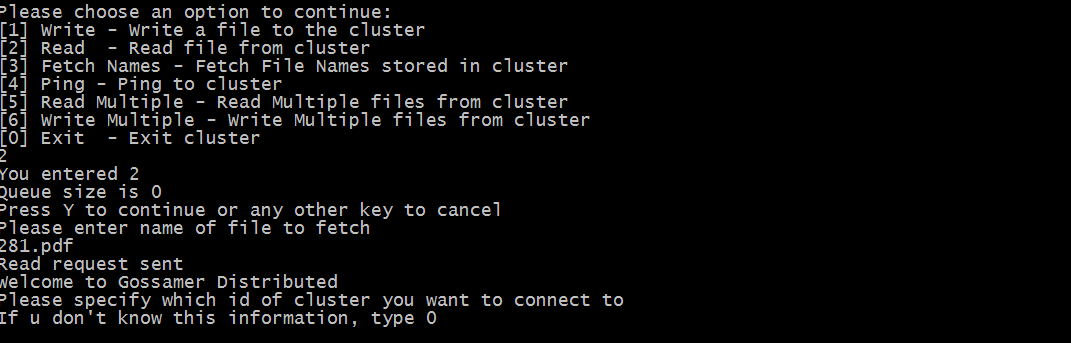


Press Enter twice to continue



Check Gossamer directory of server C:\ to see file, or check MySQL server to confirm chunks or check redis to see metadata.

Same goes for read



Check ClientStuff directory in C:\ for confirmation that chunks have been received and file created.

# References

1. <https://netty.io/>
2. https://developers.google.com/protocol-buffers/
3. <https://www.tutorialspoint.com/redis/redis_overview.htm>
4. <https://redis.io/>
5. <http://www.ianswer4u.com/2011/05/mesh-topology-advantages-and.html>
6. <https://raft.github.io/>

# Contributions

1. **Ashutosh Singh**: Node discovery, Client API, Queuing at Client, Write by multithreading, Saving metadata to Redis, Write to leader by messages, Read from leader, Refactoring, Research, Testing
2. **Viraj Nilakh**: Dynamic node addition, Dynamic node deletion, Node discovery, Raft Leader election, Work Stealing, Queuing on Server, State design pattern, Testing, Research, Global communications.
3. **Shruti Padmanabhan**: Read/write to MySQL, store chunks into MySQL, Sending data to a particular node, replication (broadcast work message), Research, Testing