HETEROGENOUS DISTRIBUTED DATA STORE

HARISH KUMAR KV |SAURABH JINTURKAR | PRASANNA GAJBHIYE | VAISHNAVI GALGALI

CMPE-275 : ENTERPISE APPLICATION DEVELOPMENT [SAN JOSE STATE UNIVERSITY]

Table of Contents

List of Figures

Introduction

Abstract

The main objective of the project is to design and implement a distributed heterogeneous data store called fluffy. The system is designed with nodes forming a cluster and clusters forming super cluster. Individual nodes had separate databases running on them which co operated with the leader in the cluster for storing and retrieving data. The client interacts with the leader in the cluster to perform storage data. The system is independent on data format and can store any incoming data format. Many clusters together formed a super cluster. The diagram of super cluster and individual cluster is as shown below.

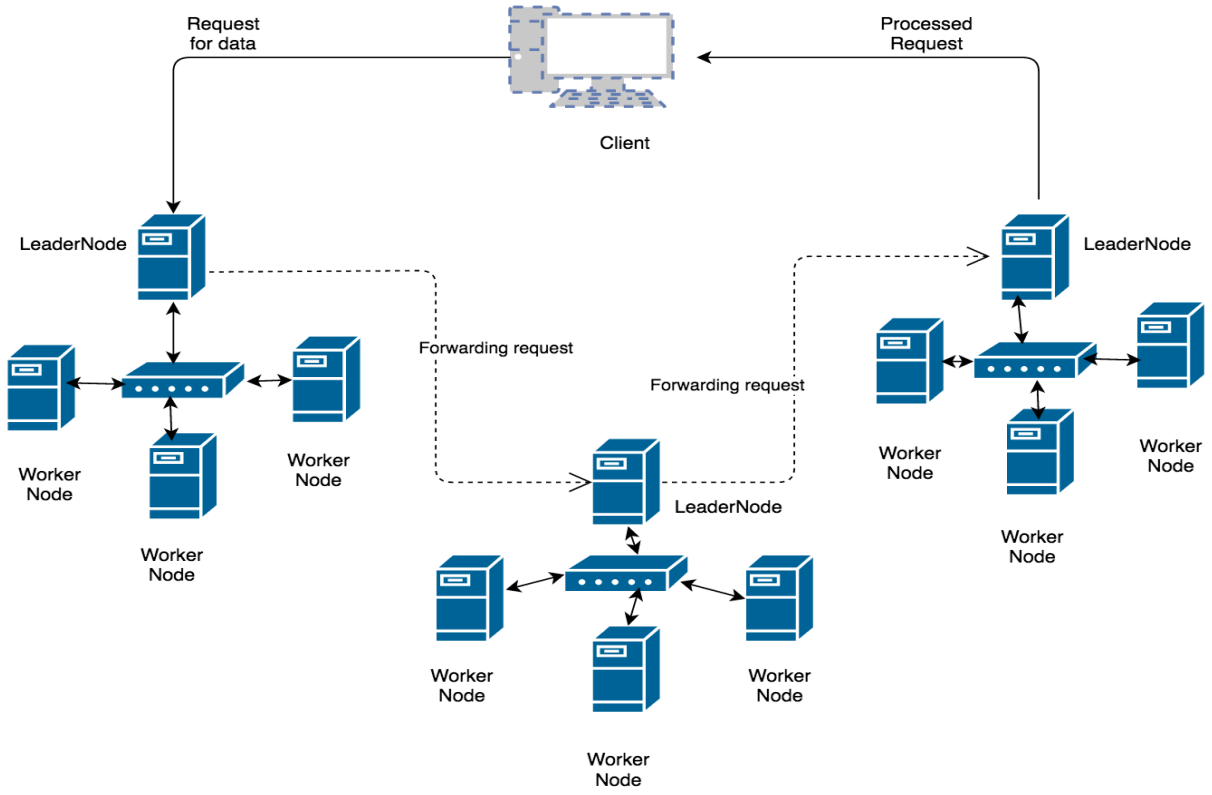


Fig. 1: Inter-cluster Communication

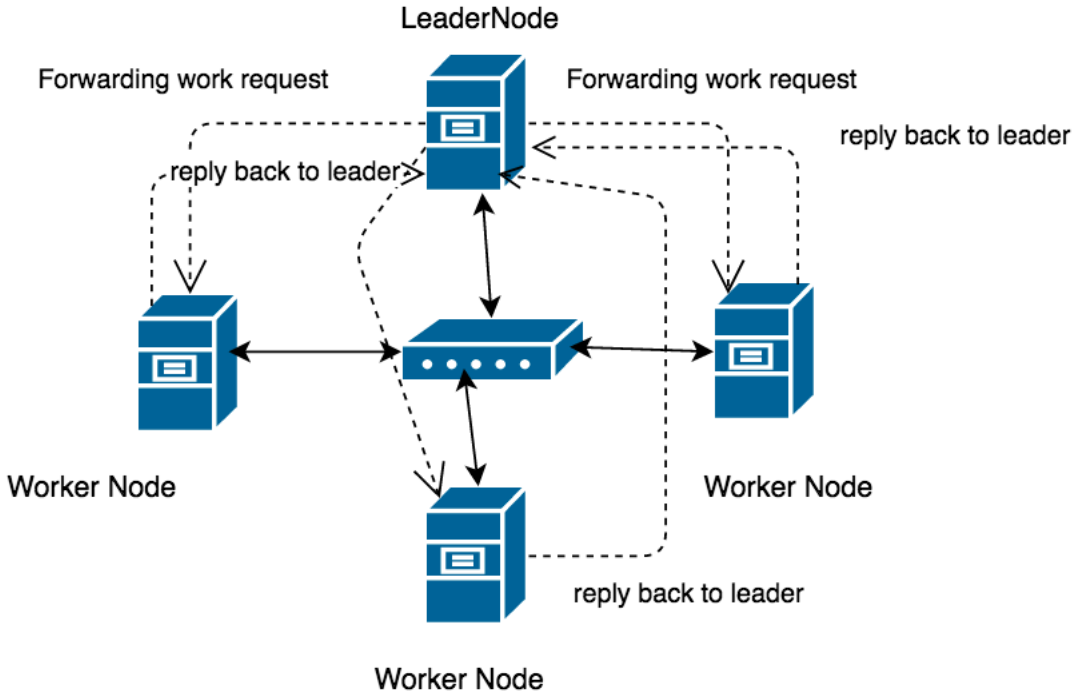


Fig 2. Intra-Cluster Communication

Problem Definition

Storing data in the cloud or on a network of servers share a common challenge -
how to securely share, store, and survive failures and corruption of data. There are many distributed data storage systems like dropbox, google drive, etc. using multiple servers and redundant solutions. The difficulties faced to store and retrieve data from a large distributed heterogeneous platform are as follows:

- **Distributed Storage:** The data chunks should be replicated in multiple nodes of the cluster to avoid single point failures. The replicated chunks should be discarded and only one copy of the data should be returned to serve "GET" request from client.
- **Single-point-of-control:** There should be only one master node to coordinate the tasks among the worker servers and client. Existence of two leaders in the network might create chaotic situations.
- **Single-point-of-failure:** As there is a single point of control in distributed system failure of the node may lead to failure of the entire system.
- **Performance of the System:** Large size messages between the nodes and the client and master node may lead to degradation of system performance.

- **Complexity due to Distributed Nature:** Distributed system introduces an additional layer of cluster and node interaction management which may lead to many complexities like maintaining the consistency of data and deadlocks among the processes interacting on the same system resource.
- **Scalability:** Synchronous communication in distributed system can hinder vertical scalability of the system while asynchronous communication may lead to data inconsistency in the system.

Proposed Solution

We aim to design and implement a system which uses Netty and Protobuf as core libraries. Netty provides a scalable framework which is good for inter-network communication while protobuf provides fast serialization of data objects. In our system, we would build up on the Netty-pipe 3 code and add new functionalities to address the issues of overlay network, data storage and data replication. Following sections discuss in depth solution.

Project Tools

Netty-3

Netty is a non-blocking input/output client-server framework for development of Java network applications. The main advantage of using netty is it an event driven asynchronous application server. Asynchronous means the process continues to work unless other process interrupt the running process with a request. Main advantages of asynchronous communication are producer and consumer need not to be synchronized and can work independently. It also improves the performance and resource utilization of the system. Netty achieved ease of development, performance, stability and flexibility.

Features of Netty:

- **Ease of use**: Netty has built in understanding of HTTP protocol so you can provide simple web services.
- **Performance**: Netty is asynchronous and non-blocking which ultimately improves system performance.
- **Scalability**: Netty approach for socket programming scales better which is important if your system needs to handle thousands of requests.
- **Security**: Netty provides complete SSL/TSL support.

Protobuf

Protobuf is a serialization format which is provided by google to send data across wire. Protocol buffers are widely used to communicate between services. Protobuf is a language-neutral, platform-neutral, and extensible way for serializing data for use in communication protocol. The most flexible, efficient and automated mechanism for serializing structured data is Protocol buffers. Each protobuf message is a small logical record of information containing series of name value pairs. Protobuf provides cross language communication unlike serialization like Java serialization.

Advantages of Protobuf over Json:

- *Schemas are awesome:* Encoding the semantics of the business object in your proto format is enough to help ensure that signal doesn't get lost between applications.
- *Backward compatibility for free:* With number fields in proto definitions it becomes easy to prevent errors and makes rolling out features and services simple.
- *Less Boiler Plate Code:* Your schema evolves with proto generated classes, leaving more time to focus on challenges of keeping your application growing and building your product.
- *Validations and Extensibility:* The "required", "optional", "repeated" are powerful keywords because they help the protocol buffers library to raise exceptions if you try to encode the object by keeping the required field blank.

- *Easy Language Interoperability:* As protocol buffers are implemented in many languages they make interoperability between application in your architecture simple.

Redis

Redis is an open source, in-memory data structure store, used as database. It supports many data structures few of which are strings, hashes, lists, sets, sorted sets, bitmaps. Redis is a data structures server which can hold complex data structures against a specified key. In this project, we have made use of hashes data structure which are maps containing of fields associated with values. For instance, when the PUT request is made by the client, the data chunk with respective sequence id is stored at the given key. We have also made use of JedisPool, a threadsafe pool of network connections rather than a lot of Jedis instances because it would have meant a lot of sockets, connections, and it resulting strange errors.

Key	**Field**	**Value**
key1	Sequence_id_0	metadata of the file/value being stored
key1	Sequence_id_1	data_chunk_0
key1	Sequence_id_2	data_chunk_1
key2	Sequence_id_0	metadata of the file/value being stored
key2	Sequence_id_1	data_chunk_0
key2	Sequence_id_2	data_chunk_1

Memcache

Memcache is an in memory store of key value pairs for small chunks of arbitrary data (strings, data). Memcache APIs provide a large hash table distributed across multiple machines. The size of this hash table is typically very large. The Memcache server maintains an associative array of key value pair. Memcache also provides an optional SASL authentication support. Memcache is simple yet powerful and its simple design helps is quick deployment, ease of development and solves many data storage problems.

Features:

- Free and open source
- High Performance
- Generic in nature
- API is available in most popular design.

LevelDB

LevelDB is key-value store written by Google. It is an open source and NOSQL database. LevelDB stores keys and values in arbitrary byte arrays. LevelDB does not support SQL queries and indexing. LevelDB is sorted on keys by default and allows complex operations like replication, map-reduce, pub-sub, etc.

Features:

- Sorted by keys
- Arbitrary byte array
- Compressed Storage
- Embeddable and Networkable

Apache Geode

Apache Geode provides database like consistency model, reliable transaction processing. It has a shared nothing architecture due to which it helps Apache geode to maintain low latency and high concurrency. Apache Geode provide that provides real-time, consistent access to data-intensive applications throughout widely distributed cloud architectures.

Features:

- Provides redundancy, replication and shared nothing architecture.
- Asynchronous as well as synchronous data update propagation.
- REST-APIs for REST-enabled application
- Persistence architecture to deliver fail-safe reliability and performance.

Implementation

To create a heterogenous data store we implemented a comman interface called IDBHandler.
The above four databases implement this common interface. We have also written unit tests for every database. Hence in future we can also add more heterogenous databases by just implementing this interface.

```java
package dbhandlers;

import java.util.List;
import java.util.Map;

/**
 * @author Saurabh
 *
 */
public interface IDBHandler {

	/**
	 * Method to store data in database with sequence id.
	 *
	 * Two tables should be maintained. One to store key to sequence id pair.
	 * Another to store data against key+sequenceId.
	 *
	 * @param key - Key as a string
	 * @param sequenceId - Sequence Id for the data chunk
	 * @param value - Data to store
	 * @return Key at which object is stored
	 */
	String put(String key, int sequenceId, byte[] value);

	/**
	 * Method to store object. Key should be calculated implicitly.<br>
	 *
	 * Note: We should not store more than one chunk of data using store. <br>
	 * <b>Should maintain entries in both tables with sequenceId as 0.</b>
	 *
	 * @param value - Object to store
	 * @return Key at which object is stored
	 */
	String store(byte[] value);


	/**
	 * Method to retrieve data stored at the key.
	 * DBHandlers should fetch all the chunks for the data with the key.
	 * <pre>Process should be as follows:
	 * 1. Fetch all the sequence IDs from table1.
	 * 2. Fetch all data chunks from the table2.
	 * 3. Return map of chunkId and Data as byte array.
	 * </pre>
	 * @param key
	 * @return
```

```java
	 * @return
	 */
	public Map<Integer, byte[]> get(String key);

	/**
	 * Update chunk of data stored at sequenceId.
	 *
	 * @param key
	 * @param sequenceId
	 * @param value
	 * @return
	 */
	public boolean update(String key, int sequenceId, byte[] value);


	/**
	 * Method to remove all data chunks at the key.
	 *
	 * @param key - key to remove
	 * @return Map of chunk id and data
	 */
	public Map<Integer, byte[]> remove(String key);

	/**
	 * Method to check if the key is present in the database.
	 *
	 * @param key
	 * @return true if key is present; otherwise false.
	 */
	public boolean hasKey(String key);

	/**
	 * Get all sequenceIds present at the node.
	 *
	 * @param key - Key
	 * @return List of sequenceIds present for the key.
	 */
	public List<Integer> getSequenceIds(String key);

	/**
	 * Utility method to know which database is used at the back end.
	 *
	 * @return
	 */
	public String getDatabaseVendor();
}
```

System Design

Server-side Architecture Diagram

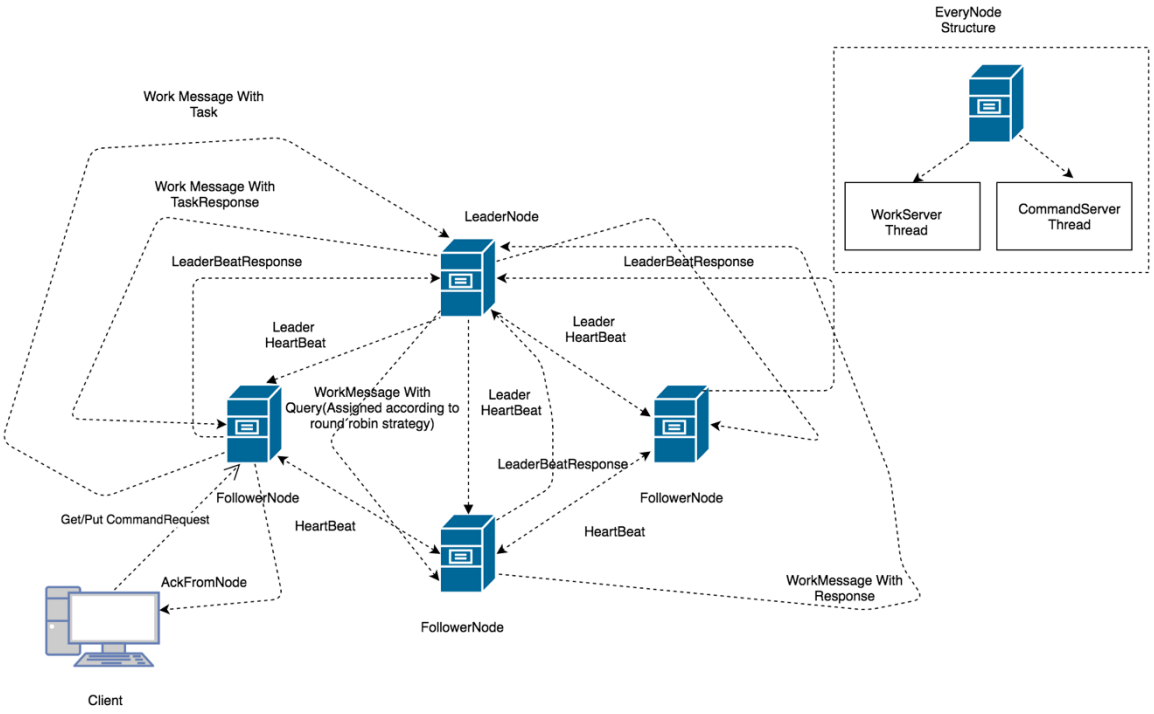


Fig. 3 Server-side Architecture

Implementation

The cluster implementation and message interaction between the nodes is as shown in the above figure. The cluster at any time consists of one leader node as well as and followers acting as slaves to leader node. The leader maintains the list of followers by sending *LeaderHeartBeat* message to all the followers. Followers reply to leader with *LeaderHeartBestResponse* message to let the leader know that it is available to serve the requests. Also the adjacent nodes send Heartbeat message to each other to maintain the EdgeInfo. Steps followed when the client makes GET/PUT request:

- Client makes GET/PUT request to any node.
- The node forwards this request to leader node.
- The leader node redirects the request to the worker node based on round robin strategy.
- The selected node serves the request and replies back to leader node.
- Leader node acknowledges the client facing node about the server request status.
- Client facing node replies back to client.

Topologies Handled:

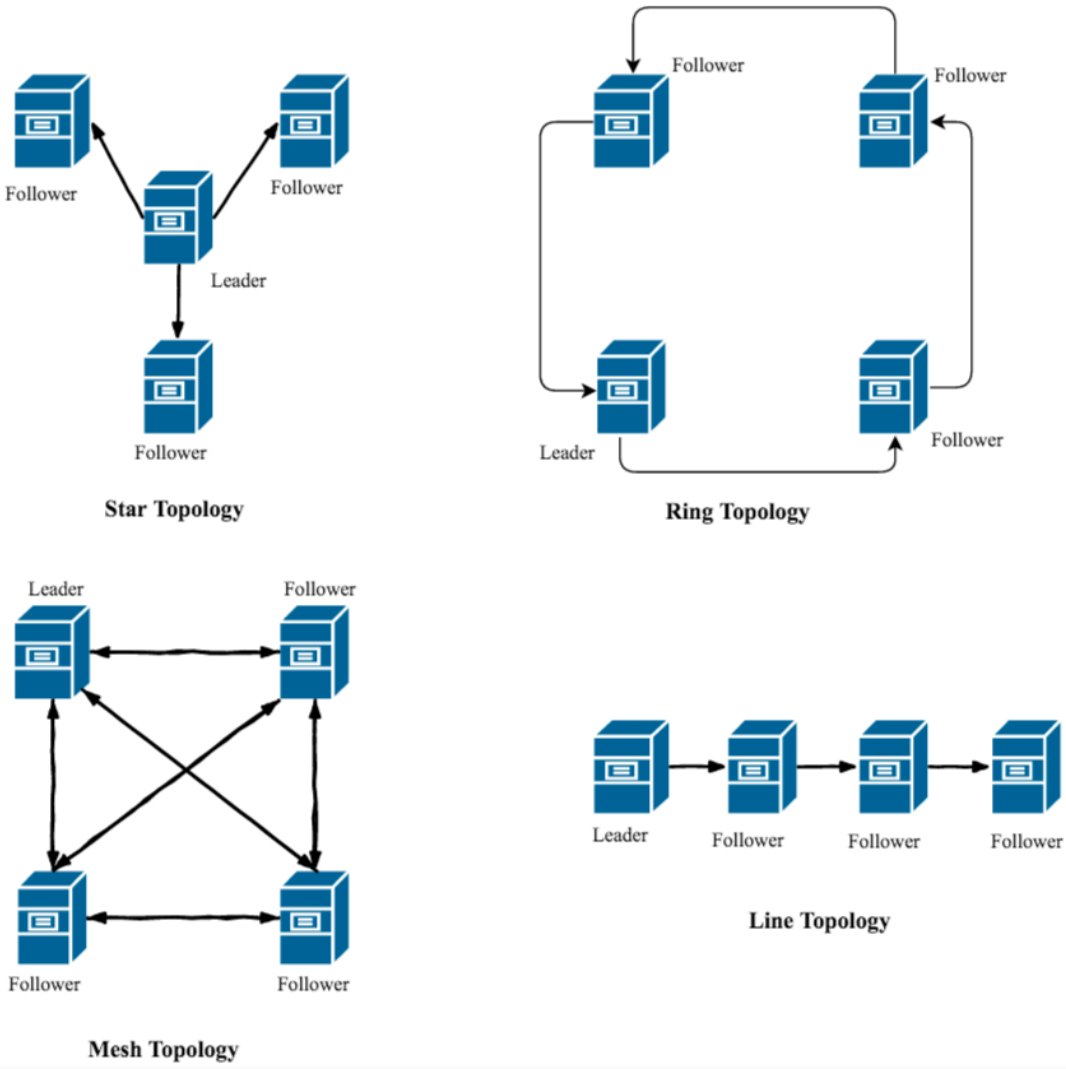


Fig. 4 Different Types of Topologies Tested

The handling of topologies is done by creating an **Overlay Network (*Router.java*).** This class has two methods *route* and *route0* which helped to us to achieve the following:

1. Minimize the number of if-else checks.
2. Reduce the number of messages flowing in the network by downsizing the max hops.
3. As we are broadcasting messages in both directions (inbound and outbound edges), it has given us the flexibility to work on any topology, like line topology.

Design Patterns Implemented:

- **State Design Pattern:**
  This design pattern is implemented to handle the states of nodes in the leader election algorithm.

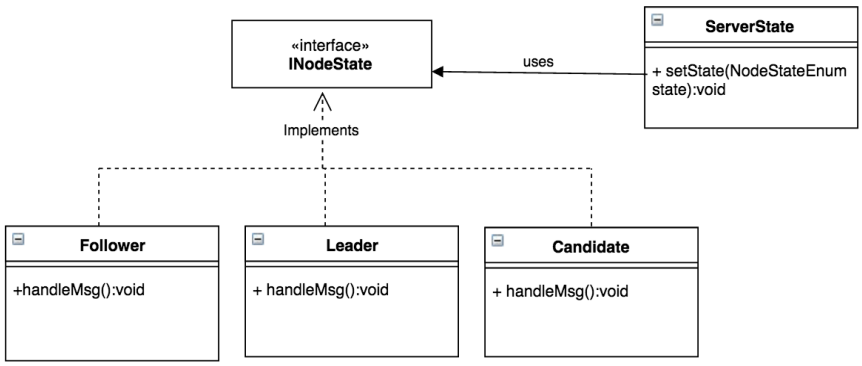


Fig. 5 Class Diagram of State Design Pattern

- **Observer Design Pattern:** This pattern is used to monitor the changes at run time in the routing configuration and notify the observers to update the configuration of the cluster.

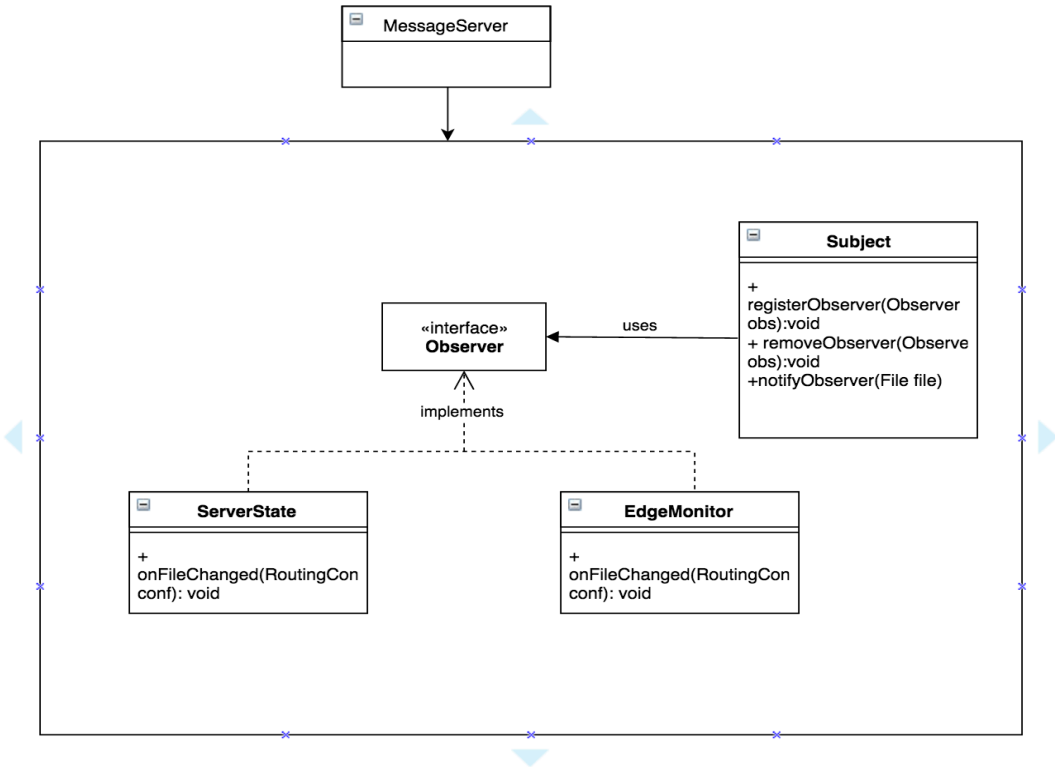


Fig. 6 Class Diagram of Observer Design Pattern

- **Chain of Responsibility:**
  The chain of responsibility design pattern is implemented to handle the different types of work messages like state message, beat message etc.

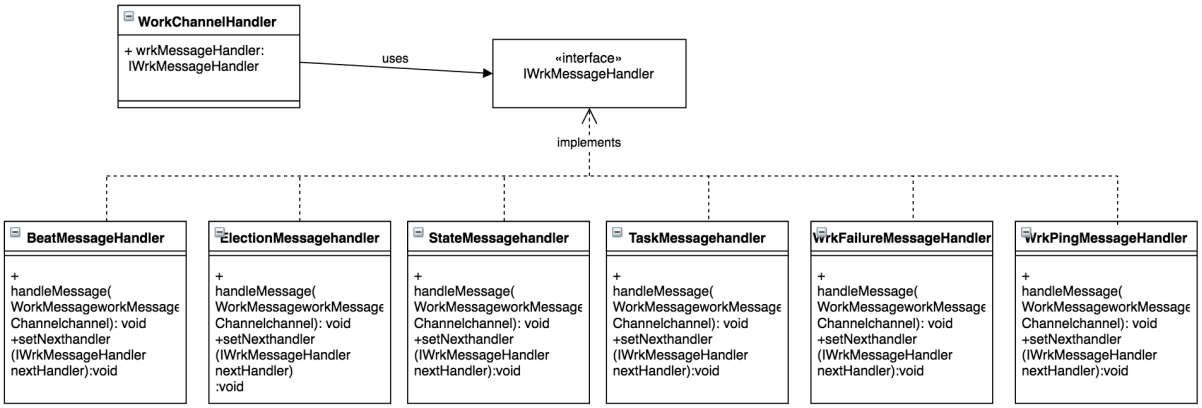


Fig. 7 Class Diagram of Chain of Responsibility Design Pattern

Leader Election

Leader election is a process of designating a single process as the organizer of some task in a distributed computing world. Each node in network will co-ordinate with the leader for every task node has to perform. Leader is election through election process which happens when ever a node identifies that there is no leader coordinating the network. Any node is eligible to become a leader.

In this project we have implemented Leader Election on the basis of Raft Algorithm. According to raft any given node can be in either of 3 states, Candidate, Follower and Leader. Raft implements algorithm by first electing a distinguished leader, then giving the leader complete responsibility for managing the entire network. This solution allow a couple of machines to work as a coherent group that can survive the failures of some of its members.

Implementation

- Implemented a state machine which defines the current state of node and decisions are taken based on the node state.
- Every node when started go to Follower State.
- Heart beat Timer will be started whenever a node is in follower state. Timer listens to leader heartbeat.

- Follower will move to Candidate state, when timer listening on leader heart beat expires.

```
W:\JDK's\jdk1.8.0_60\bin\java ...
6
[main] INFO server - in message server
[main] INFO Leader Health Monitor - ~~~~~~~~~Follower - Started Leader Monitor
[main] INFO Timer - ******** Follower - Election Timer, Request to start timer: main
[main] INFO task - Added Observers
[main] INFO task - started monitoring
[Thread-2] INFO Timer - ********Timer started: Thread-2
```

- Election Timer will be started when node moves to Candidate State.

```
[Thread-2] INFO Follower - ******ELECTION TIMED OUT******
[Thread-2] INFO Follower - ~~~~~~~~~Follower - Before State Change
[Thread-2] INFO Timer - ********Follower - Election Timer, Request to cancel timer: Thread-2
[Thread-2] INFO Candidate - ~~~~~~~~~Candidate - After State Change Event
[Thread-2] INFO Timer - ******** Cand - Election Timer, Request to start timer: Thread-2
[Thread-17] INFO Timer - ********Timer started: Thread-17
[Thread-17] INFO Timer - ********Timed out: Thread-17
[Thread-17] INFO Candidate - ~~~~~~~~~Candidate - Notify Timeout Event - Thread-17
[Thread-17] INFO Timer - ********Cand - Election Timer, Request to cancel timer: Thread-17
[Thread-17] INFO Candidate - ################################
[Thread-17] INFO Candidate - Size of the network is: 1
[Thread-17] INFO Candidate - Required vote count is: 1
[Thread-17] INFO Candidate - Time: 1460175249466
[Thread-17] INFO Candidate - ################################
[Thread-17] INFO Timer - ******** Cand - Election Timer, Request to start timer: Thread-17
[Thread-18] INFO Timer - ********Timer started: Thread-18
[Thread-18] INFO Timer - ********Timed out: Thread-18
[Thread-18] INFO Candidate - ##########################
[Thread-18] INFO Candidate - Election is over..
```

- Candidate will start an election on election timeout.
    - First Step of election is to broadcast GETCLUSTERSIZE message, to identify number of nodes active in the network and Candidate starts a timer waiting for reply.

```
[java] [nioEventLoopGroup-2-6] INFO Print Util - header {
[java]   node_id: 2
[java]   time: 1460173628086
[java]   destination: -1
[java]   max_hops: 0
[java] }
[java] secret: 1
[java] leader {
[java]   action: GETCLUSTERSIZE
[java] }
```

    - Each node reply with SIZEIS message. In the Current implementation each node will reply with its own Id to the Candidate. Even Candidate's reply to this message because here intention is to identify the number of active nodes in network.

```
[java] [nioEventLoopGroup-2-6] INFO Router - MAX HOPS is Zero! Dropping message...
[java] [nioEventLoopGroup-2-6] INFO Work Channel Handler - No need to handle message..
[java] [nioEventLoopGroup-2-6] INFO Print Util - Election
[java] [nioEventLoopGroup-2-6] INFO Print Util - header {
[java]   node_id: 6
[java]   time: 1460173628094
[java]   destination: 2
[java]   max_hops: 5
[java] }
[java] secret: 1
[java] leader {
[java]   action: SIZEIS
[java] }
```

- Once the timer started in first step expires, Candidate will update increment his Election Term and broadcast VOTEREQUEST to all nodes in the network and starts a ***vote response*** timer.

```
[java] [nioEventLoopGroup-4-4] INFO Print Util - header {
[java]   node_id: 6
[java]   time: 1460173630486
[java]   destination: -1
[java]   max_hops: 4
[java] }
[java] secret: 1
[java] leader {
[java]   action: VOTEREQUEST
[java]   leader_id: 6
[java]   election_id: 2
[java] }
```

- Nodes in the follower state will reply to the candidate.
- Nodes in the Candidate state will compare with their election term. If Election term in VOTEREQUEST is higher node will go to Follower state, if not then drop the message.
- Once the ***vote response*** timer expires then Candidate count the number of votes he got. If Candidate received the majority based on the cluster size Candidate know then Candidate will become a Leader and broadcast LEADERIS message in the network.

```
[java] [nioEventLoopGroup-4-4] INFO Print Util - header {
[java]   node_id: 6
[java]   time: 1460173634490
[java]   destination: -1
[java]   max_hops: 4
[java] }
[java] secret: 1
[java] leader {
[java]   action: THELEADERIS
[java]   state: LEADERALIVE
[java]   leader_id: 6
[java]   election_id: 2
[java] }
```

- If Candidate did not receive maximum number of votes then Candidate will go to random timeout and re start election. This process continues.

- Once the Leader is elected, Leader will broadcast LEADERIS message to all the nodes and starts a task.
- Task will take care of sending Leader Heart Beat to all the followers and also Leader will keep track of all the followers responding to the leader heartbeat.

```
[java] [nioEventLoopGroup-4-4] INFO Print Util - header {
[java]   node_id: 6
[java]   time: 1460173634490
[java]   destination: -1
[java]   max_hops: 4
[java] }
[java] secret: 1
[java] leader {
[java]   action: BEAT
[java]   state: LEADERALIVE
[java]   leader_id: 6
[java]   election_id: 2
[java] }
```

Code snippets

- **Code to construct State Objects.**
  - Existing Server state class is considered as Context which holds the current state of a node.

```
public ServerState(RoutingConf conf) {
  this.conf = conf;
  leader = new Leader(this);
  candidate = new Candidate(this);
  follower = new Follower(this);
  currentState = follower;
  leaderId = new AtomicInteger(-1);
  leaderHeartBeatdt = new AtomicLong(Long.MAX_VALUE);
  // To ensure that I will wait for heart beat timeout
  electionId = new AtomicInteger(0);
  votedFor = new AtomicInteger(-1);
}
```

- **Timer**
  - Responsibility of this timer is to wait for some time and notify the owner who created the timer.
  - Object Owner should implement TimeoutListener Interface.
  - This timer provides functionality to cancel, restart and restart with different listener and timeout.

```
public class Timer {

  private static final Logger logger = LoggerFactory.getLogger ("Timer");
  private static final boolean debug = true;
  private final String identifier;
  private long timeout;
  private TimeoutListener listener;
  private TimerThread timerThread;

  public Timer(TimeoutListener listener, long timeout, String identifier) {
    this.listener = listener;
    this.timeout = timeout;
    this.identifier = identifier;
  }

  public void start()  {
    if(debug)
      logger.info ("******** " + identifier+ ", Request to start timer: " + Thread.currentThr

    timerThread = new TimerThread ();
    timerThread.start ();
```

```
    }

    public void start(long timeout) {
        this.timeout = timeout;
        start ();
    }

    public void start(TimeoutListener listener, long timeout) {
        this.listener = listener;
        this.timeout = timeout;
        start ();
    }

    public void cancel()    {
        if(timerThread == null) {
            return;
        }
        if(debug)
            logger.info ("********" + identifier + ", Request to cancel timer: " + Thread.currentT

        timerThread.interrupt ();
    }
    private class TimerThread extends Thread{

        @Override
        public void run(){
            try {
                if(debug)
                    logger.info ("********Timer started: " + Thread.currentThread ().getName ());

                synchronized (this) {
                    wait(timeout);
                }

                if(debug)
                    logger.info ("********Timed out: " + Thread.currentThread ().getName ());

                listener.notifyTimeout ();

            } catch (InterruptedException e) {
                logger.info ("********Timer was interrupted: " + Thread.currentThread ().getName
            }
        }
    }
}
```

- **INodeState**
  - All the states should implement this interface.
  - API's supported by this interface is as shown below:

```
public interface INodeState {

    void beforeStateChange();

    void afterStateChange();

    void handleGetClusterSize(WorkMessage workMessage, Channel channel);

    void handleSizeIs(WorkMessage workMessage, Channel channel);

    void handleLeaderIs(WorkMessage workMessage, Channel channel);

    void handleVoteRequest(WorkMessage workMessage, Channel channel);

    void handleVoteResponse(WorkMessage workMessage, Channel channel);

    void handleWhoIsTheLeader(WorkMessage workMessage, Channel channel);

    void handleBeat(WorkMessage workMessage, Channel channel);
```

```
void handleCmdQuery(WorkMessage workMessage, Channel channel);

void handleCmdResponse(WorkMessage workMessage, Channel channel);

void handleCmdError(WorkMessage workMessage, Channel channel);

default void handleMessage(WorkMessage workMessage, Channel channel) {
    LeaderStatus leaderStatus = workMessage.getLeader();
    switch (leaderStatus.getAction()) {
        case WHOISTHELEADER:
            handleWhoIsTheLeader(workMessage, channel);
            break;
        case THELEADERIS:
            handleLeaderIs(workMessage, channel);
            break;
        case GETCLUSTERSIZE:
            handleGetClusterSize(workMessage, channel);
            break;
        case SIZEIS:
            handleSizeIs(workMessage, channel);
            break;
        case VOTEREQUEST:
            handleVoteRequest(workMessage, channel);
            break;
        case VOTERESPONSE:
            handleVoteResponse(workMessage, channel);
            break;
        case BEAT:
            handleBeat (workMessage, channel);
        default:
    }
  }
}
```

- **TimeoutListener**
  - In order to use timer defined above, classes have to implement this interface.
  - Classes implementing this interface can have timer object and register themselves to the timer to get notification after timeout.

```
public interface TimeoutListener {

    void notifyTimeout();
}
```

- **LeaderHealthMonitor**
  - This class is responsible for monitoring the health of leader. Follower will create object of this class initially and later uses to reset the timer whenever a leader is identified.
  - This monitor will notify the follower whenever monitor don't see leader heart beat in specified timeout.
  - Follower will have reference to Leader Monitor and update the time of heartbeat from leader which is monitored by task inside Leader Health Monitor.

```
public class LeaderHealthMonitor {

    private final LeaderHealthListener healthListener;
    private boolean debug = false;
    private final Logger logger = LoggerFactory.getLogger("Leader Health Monitor");
    private AtomicBoolean stop;
    private HealthMonitorTask task;
    private AtomicLong beatTime;
    private final long timeout;

    public LeaderHealthMonitor(LeaderHealthListener healthListener, long timeout) {
        this.healthListener = healthListener;
        this.timeout = timeout;
        task = new HealthMonitorTask();
        stop = new AtomicBoolean(false);
        beatTime = new AtomicLong(Long.MAX_VALUE);
    }

    /* This method will be called by the owner of Health Monitor to update the beat time of leader
     * Internal task will be monitoring this time
     * This time can be set to Long.MAX_VALUE if this thread has to be alive instead of creating
     * montor multiple times..*/
    public void onBeat(long beatTime) {
        this.beatTime.getAndSet(beatTime);
    }

    private class HealthMonitorTask extends Thread {

        @Override
        public void run() {
            try {
                while (!stop.get()) {
                    if (debug)
                        logger.info("********Started: " + new Date(System.currentTimeMillis()));

                    long currentTime = System.currentTimeMillis();

                    if ((currentTime - beatTime.get()) > timeout) {
                        healthListener.onLeaderBadHealth();
                    }
                    synchronized (this) {
                        wait((long) (timeout * 0.9));
                    }
                }
            } catch (InterruptedException e) {
                logger.info("********Timer was interrupted: " + new Date(System.currentTimeMillis
            }
        }
    }
}
```

- **LeaderHealthListener**
  - This is an interface that should be implemented by classes to get notification when there is no leader heartbeat.
  - API's supported are shown below:

```
public interface LeaderHealthListener {
    /*
     * implement this method if you are interested in the notification on leader bad health*/
    void onLeaderBadHealth();
}
```

- **FollowerListener**
  - This is an interface that should be implemented by classes to get notification about follower heartbeat. This is mainly implemented by leader who will send heart beats and will listen on the responses from the follower.
  - API's supported in this interface are as below:

```
public interface FollowerListener {

    /*
     * followerId - This will be same as Node-Id
     * */
    void addFollower(int followerId);

    void removeFollower(int followerId);
}
```

- **FollowerHealthMonitor**
  - Monitor that has an internal task that keeps track of health of each follower.
  - If any follower don't respond to Leader Heart Beat, then follower is removed from the list of followers maintained by leader.
  - Leader has to implement Follower Listener interface to get notification on bad health of follower.

```
 *
 *       This class is mainly for Leader to keep track of all the followers
 *       health status..
 *
 *       This task, when started broad cast heart messages to the follower and
 *       starts listening to the reply.
 *
 *       On reply from follower's I maintain follower2BeatTimeMap which will be
 *       updated with time beat was received mapped to the follower who sent
 *       beat.
 *
 *       Thread will be invoked in frequent intervals of time and checks if
 *       there is any follower who didn't reply, if yes notify FollowerListener
 *       to remove the node. Once the iteration is done, I again send broad
 *       cast messages to all my followers and go back to wait.
 *
 */
public class FollowerHealthMonitor {

    private boolean debug = false;
    private final Logger logger = LoggerFactory.getLogger("Follower Health Monitor");
    private ConcurrentHashMap<Integer, Long> follower2BeatTimeMap;
    private final ServerState state;
    private FollowerListener followerListener;
    private HealthMonitorTask task;
    private AtomicBoolean stop;
    private final long timeout;

    public FollowerHealthMonitor(FollowerListener followerListener, ServerState state, long timeout) {
        this.followerListener = followerListener;
        this.state = state;
        // this.interval = timeout - (long)(0.1 * timeout); // 10% lesser than
        // the original timeout
        this.timeout = timeout;
        task = new HealthMonitorTask();
        stop = new AtomicBoolean(false);
        this.follower2BeatTimeMap = new ConcurrentHashMap<>();
    }

    public void onBeat(int followerId, long heartBeatTime) { follower2BeatTimeMap.put(followerId
```

```
public void start() {
   // Broadcast heartbeat to all the followers
   LeaderStatusMessage beat = new LeaderStatusMessage (state.getConf().getNodeId());
   beat.setElectionId (state.getElectionId ());
   beat.setLeaderId (state.getLeaderId ());
   beat.setMaxHops (state.getConf ().getMaxHops ());
   beat.setLeaderAction(LeaderQuery.BEAT);
   beat.setLeaderState(LeaderState.LEADERALIVE);

   state.getEmon().broadcastMessage(beat.getMessage());

   stop.getAndSet (false);
   task.start();
}

public void cancel() {
   stop.getAndSet(true);
   task = new HealthMonitorTask ();
}

private class HealthMonitorTask extends Thread {

   private boolean broadCastBeat = false;

   @Override
   public void run() {
      try {
         while (!stop.get()) {

            if (debug)
               logger.info("********Started: " + new Date(System.currentTimeMillis()));

            if (broadCastBeat) {
               // Broadcast heartbeat to all the followers
               logger.info ("#####Leader broadcasting heartbeat to all followers");

               LeaderStatusMessage beat = new LeaderStatusMessage (state.getConf().getNodeId(
               beat.setElectionId (state.getElectionId ());
               beat.setLeaderId (state.getLeaderId ());
               beat.setMaxHops (state.getConf ().getMaxHops ());
               beat.setLeaderAction(LeaderQuery.BEAT);
               beat.setLeaderState(LeaderState.LEADERALIVE);

               state.getEmon().broadcastMessage(beat.getMessage());
               broadCastBeat = false;
               synchronized (this) {
                  wait((long)(timeout * 0.9));
               }
            } else {
               long currentTime = System.currentTimeMillis();

               ArrayList<Integer> nodes2Remove = new ArrayList<> ();

               nodes2Remove.addAll (follower2BeatTimeMap.entrySet().stream()
                     .filter(entry -> currentTime - entry.getValue() > timeout).map (Map.Entry::ge
                     .collect(Collectors.toList()));

               for(Integer nodeId : nodes2Remove)  {
                  follower2BeatTimeMap.remove (nodeId);
                  followerListener.removeFollower (nodeId);
               }

               logger.info ("####Follower heartbeats:" + follower2BeatTimeMap);
               broadCastBeat = true;
            }
         }
      } catch (InterruptedException e) {
```

Handling of LEADERIS message in Different state's:

- **Follower**

```
/*
* I check if I got this message from a new leader with new terms
* */
   if (incomingTerm >  currentTerm && inComingLeader != myLeader) {
      logger.info("LEADER IS: " + workMessage.getLeader().getLeaderId());
      state.setElectionId(incomingTerm);
      state.setLeaderId(inComingLeader);

      /*Once I get a new leader, I will cancel my previous leader monitor task and start it again for
       * new monitor*/
      long currentTime = System.currentTimeMillis ();

      state.setLeaderHeartBeatdt (currentTime);
      leaderMonitor.onBeat (currentTime); //Updating the beat time of leader once I receive Hear
   }
```

- **Candidate**

```
if (workMessage.getLeader().getElectionId() >= state.getElectionId()) {
   logger.info("LEADER IS: " + workMessage.getLeader().getLeaderId());
   state.setElectionId(workMessage.getLeader().getElectionId());
   state.setLeaderId(workMessage.getLeader().getLeaderId());

   //Cancel if there is any timer that is currently started...
   timer.cancel ();

   state.setLeaderHeartBeatdt (System.currentTimeMillis ());

   //Change the state to Follower..
   state.setState(NodeStateEnum.FOLLOWER);
}
```

- **Leader**

```
/*
* If there is another node which is leader in new term, then I update myself and go back to election state
* */
if (inComingTerm > currentTerm) {
   logger.info("LEADER IS: " + workMessage.getLeader().getLeaderId());
   state.setElectionId(workMessage.getLeader().getElectionId());
   state.setLeaderId(workMessage.getLeader().getLeaderId());
   state.setLeaderHeartBeatdt (System.currentTimeMillis ());
   state.setState(NodeStateEnum.FOLLOWER);
}

/*
* If there is another leader which is elected in same term then I go back to candidate state to start new election
* This scenario might happen because we dont have constant cluster size and we evaluate dynamically and it need no
* that in 2 seconds we get entire cluster size and split votes might occur, to be on the safe side its better to go to CAN
* */
if(inComingTerm == currentTerm) {
   state.setState (NodeStateEnum.CANDIDATE);
}
```

Handling of Beat Message(Leader) in different states:

- **Follower**

```
if(inComingTerm > currentTerm && newLeaderId != currentLeaderId) {
    state.setElectionId (inComingTerm);
    state.setLeaderId (newLeaderId);
    // Reset Leader Monitor Task
    //leaderMonitor.cancel ();

    long currentTime = System.currentTimeMillis ();

    state.setLeaderHeartBeatdt (currentTime);
    leaderMonitor.onBeat (currentTime); //Updating the beat time of leader once I receive Heart E
    //leaderMonitor.start ();

    //Return the response..
    LeaderStatusMessage leaderBeatResponse = new LeaderStatusMessage (state.getConf ().getN
    leaderBeatResponse.setLeaderId (newLeaderId);
    leaderBeatResponse.setDestination (newLeaderId);
    leaderBeatResponse.setMaxHops (state.getConf ().getMaxHops ());
    leaderBeatResponse.setLeaderAction (Election.LeaderStatus.LeaderQuery.BEAT);
    leaderBeatResponse.setLeaderState (Election.LeaderStatus.LeaderState.LEADERALIVE);

    //Write it back to the channel from where I received it...
    channel.writeAndFlush (leaderBeatResponse.getMessage ());
    //Also update other nodes in outbound about the leader heart beat..
    state.getEmon ().broadCastOutBound (leaderBeatResponse.getMessage ());
    return;
}

// I check for -1 because, If I go down and come up then I wont be having any leader...
if(inComingTerm == currentTerm &&
        (newLeaderId == currentLeaderId || currentLeaderId == -1))    {

    if(currentLeaderId ==  -1)  {
        state.setLeaderId (newLeaderId);
    }

    long currentTime = System.currentTimeMillis ();

    // Updating heartbeat..
    state.setLeaderHeartBeatdt (currentTime);
    leaderMonitor.onBeat(currentTime);

    //Return the response..
    LeaderStatusMessage leaderBeatResponse = new LeaderStatusMessage (state.getConf ().getN
    leaderBeatResponse.setLeaderId (newLeaderId);
    leaderBeatResponse.setDestination (newLeaderId);
    leaderBeatResponse.setMaxHops (state.getConf ().getMaxHops ());
    leaderBeatResponse.setLeaderAction (Election.LeaderStatus.LeaderQuery.BEAT);
    leaderBeatResponse.setLeaderState (Election.LeaderStatus.LeaderState.LEADERALIVE);

    //Write it back to the channel from where I received it...
    channel.writeAndFlush (leaderBeatResponse.getMessage ());

    //Also update other nodes in outbound about the leader heart beat..
    state.getEmon ().broadCastOutBound (leaderBeatResponse.getMessage ());
}
// TODO: Update visited nodes map
}
```

- **Candidate**

```
if (workMessage.getLeader().getElectionId() >= state.getElectionId()) {
    logger.info("*****Heart Beat From Leader: " + workMessage.getLeader().getLeaderId());
    state.setElectionId(workMessage.getLeader().getElectionId());
    state.setLeaderId(workMessage.getLeader().getLeaderId());

    //Cancel if there is any timer that is currently started...
    timer.cancel ();

    //Update the Leader Heart Beat in my Server State
    state.setLeaderHeartBeatdt (System.currentTimeMillis ());

    //Change the state to Follower..
    state.setState(NodeStateEnum.FOLLOWER);
}
```

- **Leader**

```
/*This should never happen, but for safety this means there is another leader with greater term and I should
go back to Follower*/
if(inComingTerm > currentTerm)   {
    state.setLeaderHeartBeatdt (System.currentTimeMillis ());
    state.setState (NodeStateEnum.FOLLOWER);
    return;
}

/*If I become a leader and find that there is another leader with equal term then I try to re start election*/
if(inComingTerm == currentTerm)   {
    state.setState (NodeStateEnum.CANDIDATE);
    return;
}

/*If not any of the cases above then it means I am getting beat response from my followers, so I
 * add them to the list and also update their beat time*/
int followerId = workMessage.getHeader ().getNodeId ();
addFollower (followerId); // Add follower Id

long currentTime = System.currentTimeMillis ();

followerMonitor.onBeat (followerId, currentTime); // notify follower monitor about heart beat
```

- **Before and After Change Event of Each State –** Before Event is mostly used for Cleaning up or releasing the resources hold by the current state. After Event is mainly used for starting timers. Code snippets shown below discuss about the resources released on before event and timers normally started on state change event.

- **Follower**

```
@Override
public void beforeStateChange() {
    logger.info("~~~~~~~~~Follower - Before State Change");
    timer.cancel();

    leaderMonitor.onBeat (Long.MAX_VALUE);
    //leaderMonitor.cancel();
}

@Override
public void afterStateChange() {
    long leaderHeartBeatDt = state.getLeaderHeartBeatdt ();

    if(leaderHeartBeatDt == Long.MAX_VALUE) {
        leaderHeartBeatDt = System.currentTimeMillis ();
    }

    // It might be the case, I came from either Candidate State or Leader State after receiving HB
    leaderMonitor.onBeat (leaderHeartBeatDt);
}
```

- **Candidate**

```
@Override
public void beforeStateChange() {
    logger.info("~~~~~~~~~Candidate - Before State Change Event");
    timer.cancel ();
}

@Override
public void afterStateChange() {
    logger.info("~~~~~~~~~Candidate - After State Change Event");
    clear();
    getClusterSize(); // Broadcasts GETCLUSTERSIZE message and starts a timer
}
```

- **Leader**

```
/* Release all the resources. In this case it is only followerMonitor */
@Override
public void beforeStateChange() {
    logger.info("~~~~~~~~~Leader - Handle Before State Change Event");
    getActiveNodes().clear ();
    followerMonitor.cancel ();
}

@Override
public void afterStateChange() {
    logger.info("~~~~~~~~~Leader - Handle After State Change Event");
    followerMonitor.start();
}
```

Handling VOTEREQUEST message in different states.

- **Follower**

```
if (workMessage.getLeader().getElectionId() > state.getElectionId()) {

    state.setElectionId (workMessage.getLeader ().getElectionId ());
    state.setLeaderId (workMessage.getLeader ().getLeaderId ());

    VoteResponse vote = new VoteResponse (state.getConf ().getNodeId (),
        workMessage.getLeader().getElectionId(),
        workMessage.getLeader().getLeaderId());

    //Forward to the destination node who requested for the vote..
    vote.setDestination (workMessage.getHeader ().getNodeId ());
    vote.setMaxHops (state.getConf ().getMaxHops ());

    state.setLeaderHeartBeatdt (System.currentTimeMillis ());

    //Reply to the person who sent request
    channel.writeAndFlush (vote.getMessage ());

    // Broadcast the message to outbound edges.
    // Because if my in bound edge is down, I am trying to reach my candidate in different path - sa
    state.getEmon().broadCastOutBound (vote.getMessage());
    state.setVotedFor (workMessage.getHeader ().getNodeId ());

    // Reset timer, so that if nobody becomes leader in near by future I can go to Candidate state.
```

```
// Reset timer, so that if nobody becomes leader in near by future I can go to Candidate state.
timer.cancel ();
logger.info("******** Restarting the timer, once I have given my vote*********");
timer.start ();
```

- **Candidate**

```
if (workMessage.getLeader().getElectionId() > state.getElectionId()) {
    VoteResponse vote = new VoteResponse (nodeId,
            workMessage.getLeader().getElectionId(),
            workMessage.getLeader().getLeaderId());

    vote.setDestination (workMessage.getHeader ().getNodeId ());
    vote.setMaxHops (state.getConf ().getMaxHops ());
    //Update the term Id I participated in
    state.setElectionId (workMessage.getLeader ().getElectionId ());
    state.setLeaderId (workMessage.getLeader ().getLeaderId ());

    //Reply to the person who sent request
    channel.writeAndFlush (vote.getMessage ());

    // Broadcast the message to outbound edges.
    // Because if my in bound edge is down, I am trying to reach my candidate in different path..
    state.getEmon().broadCastOutBound (vote.getMessage());

    state.setVotedFor (workMessage.getHeader ().getNodeId ());
    state.setLeaderHeartBeatdt (System.currentTimeMillis ());
    state.setState (NodeStateEnum.FOLLOWER);
}
```

- **Leader**

```
if (inComingTerm > currentTerm) {
    state.setElectionId (workMessage.getLeader ().getElectionId ());
    state.setLeaderId (workMessage.getLeader ().getLeaderId ());

    VoteResponse vote = new VoteResponse (nodeId,
            workMessage.getLeader().getElectionId(),
            workMessage.getLeader().getLeaderId());

    vote.setDestination (workMessage.getHeader ().getNodeId ());
    vote.setMaxHops (state.getConf ().getMaxHops ());

    //Reply to the person who sent request
    channel.writeAndFlush (vote.getMessage ());

    // Broadcast the message to outbound edges.
    // Because if my in bound edge is down, I am trying to reach my candidate in different path..
    state.getEmon().broadCastOutBound (vote.getMessage());
    state.setVotedFor (workMessage.getHeader ().getNodeId ());

    state.setLeaderHeartBeatdt (System.currentTimeMillis ());
    state.setState (NodeStateEnum.FOLLOWER);
}
```

Handling VOTERESPONSE message in different states.

- **Follower –** Follower don't request or response to any votes.
- **Candidate**

```
@Override
public void handleVoteResponse(WorkMessage workMessage, Channel channel) {
    logger.info("~~~~~~~~Candidate - Handle Vote Response Event");
    logger.info("Receiving Vote Response from :" + workMessage.getHeader().getNodeId());

    LeaderStatus leader = workMessage.getLeader();
    /*getLeaderId - nothing other than candiate Id, I set this value before I send vote request*/
    if (leader.getVotedFor() == state.getConf ().getNodeId () && leader.getVoteGranted()) {
        votes.put(workMessage.getHeader().getNodeId(), theObject);
        logger.info("Votes: " + votes.toString());
    }

}
```

- **Leader –** Messages coming to leader are dropped.

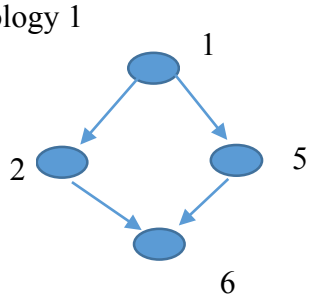
- **Series of Events happening when Candidate received the size of Entire Cluster**
  - First I cancel timer which was started after sending GETCLUSTERSIZE message.
  - Then Start election.
  - Start timer.
  - Validate number of votes, if majority become a leader, if lesser votes then start random timer.

```
logger.info("~~~~~~~~Candidate - Notify Timeout Event - " + Thread.currentT
//Cancel the previous timer...
timer.cancel ();

sizeOfCluster = visitedNodes.size() + 1;
requiredVotes = Math.round((sizeOfCluster / 2) + 0.5f);
logger.info("################################");
logger.info("Size of the network is: " + sizeOfCluster);
logger.info("Required vote count is: " + requiredVotes);
logger.info("Time: " + System.currentTimeMillis());
logger.info("################################");

startElection();

//start a new timer...
timer.start (() -> {
    int myVoteCount = votes.size () + 1;
    logger.info("############################");
    logger.info("Election is over..");
    logger.info("Time: " + System.currentTimeMillis());

    if (myVoteCount > requiredVotes ||
          (myVoteCount == 1 && requiredVotes == 1)) {
       /*Update myself as a Leader in the new term and broad cast LEADERIS message*/
       state.setElectionId(state.getElectionId () + 1);
       state.setLeaderId (state.getConf ().getNodeId ());

       //TODO this should be in separate method.
       state.getEmon().broadcastMessage(util
             .createLeaderIsMessage(state));
       logger.info("State is leader now..");
       state.setState(NodeStateEnum.LEADER);
    }else    {
       timer.start (this, 150 + new Random ().nextInt (150));
    }

    clear();
}, state.getConf ().getElectionTimeout ());
```

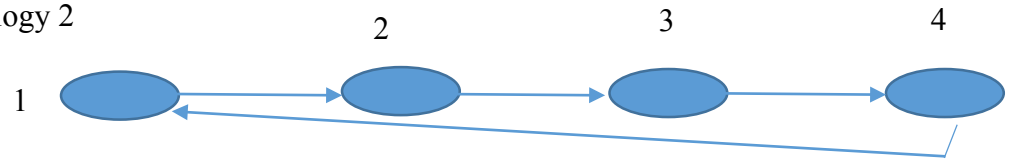
Test Cases tested and passed

Let us consider 4 nodes and topologies defined in the diagram below:

Topology 1



Topology 2



Topology 3

Network a **Network b**

Topology 1:

a. All 4 nodes were up. Leader election happened and node #1 became leader with term Id 1.
b. 2, 5 & 6 were the followers.
c. Node 1 was removed from the network, re-election took place and leader was elected among 2, 5& 6. In our testing it was 2 and term id was 2.
d. Node 2 was also removed from the network.
e. Node 3 became leader with term Id 3.
f. Node 1 and Node 2 was connected to network again and they both became followers to Node 3 because Node 3 is leader for term 3 and Node 1 is having term 1 and Node 2 is having term 2.

Topology #2

a. All 4 nodes were up. Leader election happened and node #2 became leader with term Id 1.
b. Routing configuration of node #1 was updated to remove connection from Node #1 to Node #2.
c. There was no leader election, because Node #1 and Node #2 has connection through Node #3 and Node #4.
d. Node #2 was totally removed and there was a re-election between 1, 3 & 4. Node #3 became leader.
e. Node #2 was connected dynamically again to the network and Node #3 was still leader because of higher term.

Topology #3

a. In this topology there were 2 network's which is Node #1 connected to Node #2 and Node #3 connected to Node #4.
b. All nodes were up. Election happened in both networks. Node #1 and Node #3 were leaders with same term.
c. Both the networks were connected. Re-election happened and Node #1 became leader.

Dynamic Topology changes:

Our project succeeded in handling dynamic changes in the routing configuration. User can change the routing and topology of the system when the cluster is live. This is implemented using Observer Pattern. We have created a thread at system startup which continuously polls the routing configurations in the cluster for every 2 milliseconds. If there are configuration changes in any route.conf the thread notifies all the observers which will be affected by configuration changes to

reload the configuration and use the updated configuration. The new cluster configuration and topology is created within fraction of seconds.

Goals Achieved:

- Dynamic changes can be made in the topology.
- Max hops values can be updated at runtime.

Code Snippets:

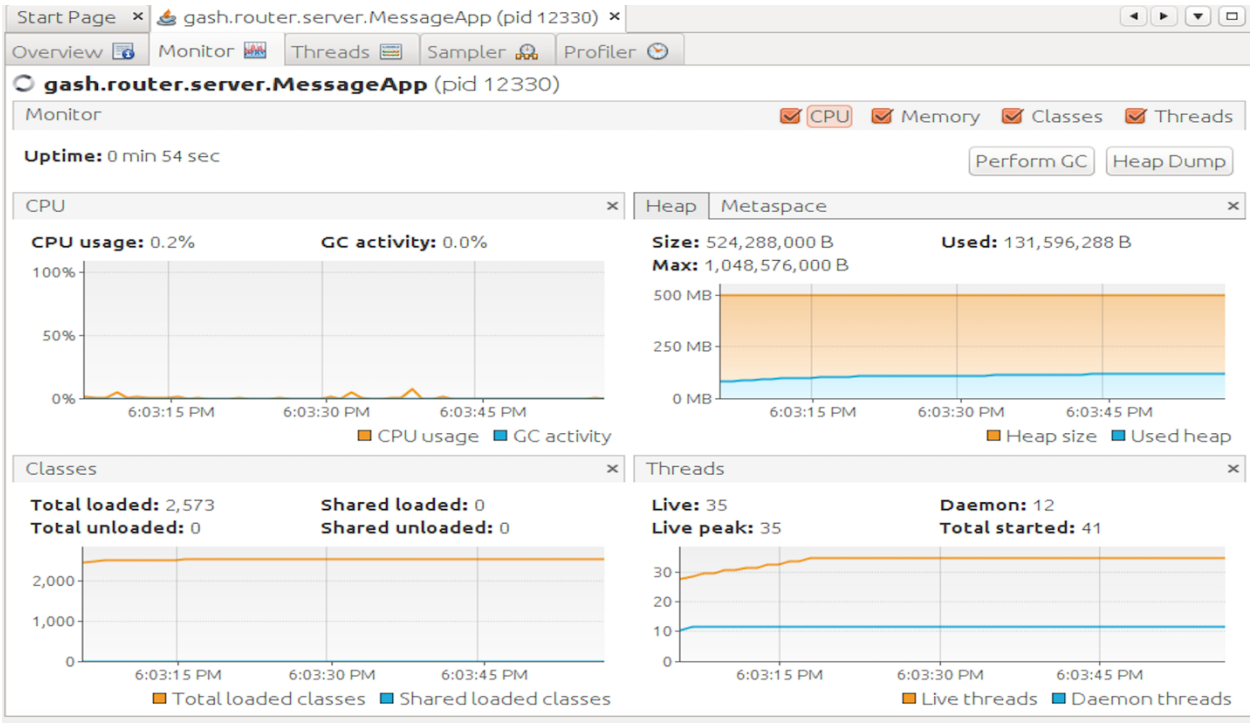
Monitoring Thread:

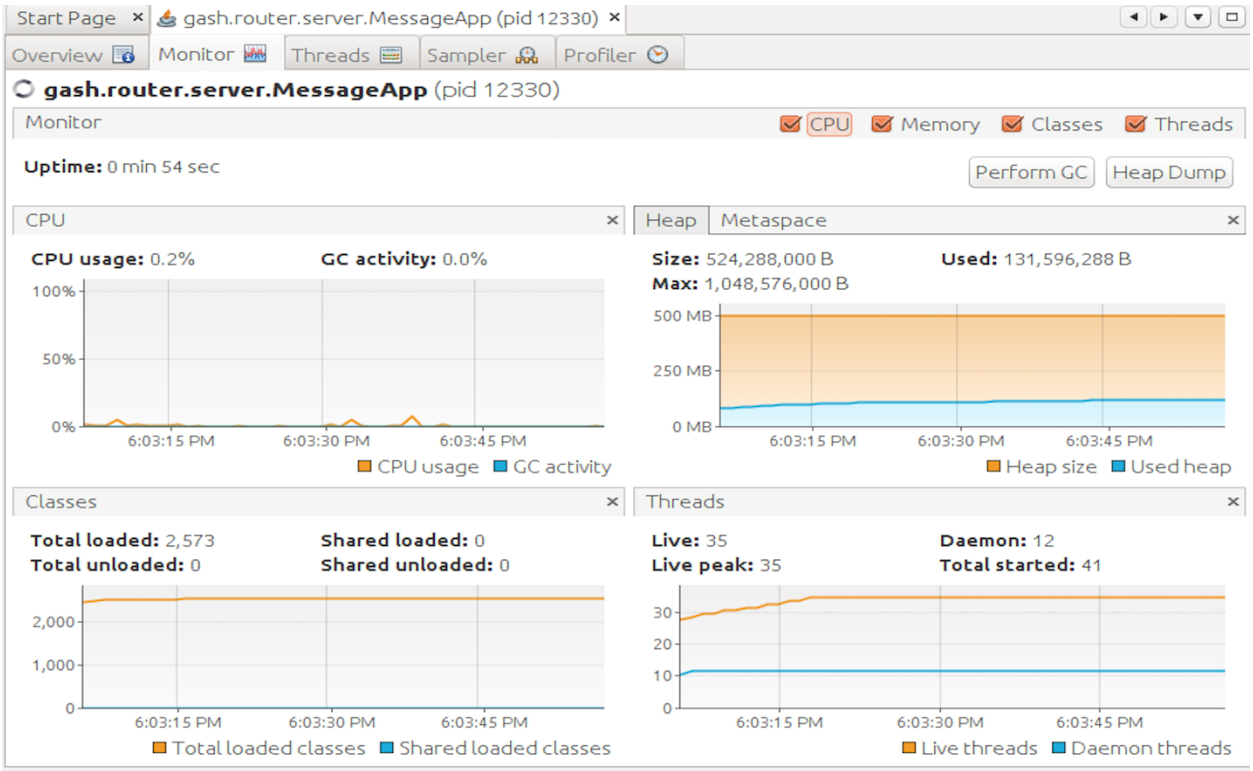
```
private class FileWatcher extends Thread {

        private AtomicBoolean stop = new AtomicBoolean(false);
        public File file;
        public FileWatcher(File file) {
                this.file = file;
        }

        @Override
        public void run() {
                try (WatchService watcher = FileSystems.getDefault().newWatchService()) {
                        Path path = file.toPath().getParent();
                        path.register(watcher, StandardWatchEventKinds.ENTRY_MODIFY);
                        while (!isStopped()) {
                                WatchKey key;
                                try {
                                        key = watcher.poll(200, TimeUnit.MILLISECONDS);
                                } catch (InterruptedException e) {
                                        return;
                                }
                                if (key == null) {
                                        Thread.yield();
                                        continue;
                                }

                                for (WatchEvent<?> event : key.pollEvents()) {
                                        WatchEvent.Kind<?> kind = event.kind();

                                        @SuppressWarnings("unchecked")
                                        WatchEvent<Path> ev = (WatchEvent<Path>) event;
                                        Path filename = ev.context();

                                        if (kind == StandardWatchEventKinds.OVERFLOW) {
                                                Thread.yield();
                                                continue;
                                        } else if (kind == java.nio.file.StandardWatchEventKinds.ENTRY_MODIFY) {
                                                logger.info("in event ");
                                                notifyObservers(file);
                                        }
                                        boolean valid = key.reset();
                                        if (!valid) {
                                                break;
                                        }
                                }
                                Thread.yield();
```

Notifying Observers:

```
public class MonitoringTask implements Subject {

	private ArrayList<Observer> observers = new ArrayList<Observer>();

	private static Logger logger = LoggerFactory.getLogger("task");



	@Override
	public void registerObserver(Observer observer) {
		logger.info("Added Observers ");
		observers.add(observer);

	}

	@Override
	public void removeObserver(Observer observer) {
		observers.remove(observer);

	}

	@Override
	public void notifyObservers(File file) {
		RoutingConf conf =init(file);
		logger.info("Notifying Observers ");

		for (Observer ob : observers) {
			ob.onFileChanged(conf);
		}

	}
```

System Performance





Data Storage

Storage.proto

We have defined a new proto file to develop storage functionality. We have defined 3 types of messages.

Following enum defines four operations which can be performed on the data.

```
/*
 * ============Data Operations============
 * There are 4 types of operations which can be performed on data. They are
 * analogous to CRUD operations in databases.
 */
enum Action {
    GET = 1;
    STORE = 2;
    UPDATE = 3;
    DELETE = 4;
}
```

We have defined metadata message which will store information about the stored data. We store sequence size so that we can successfully reconstruct complete file/data from chunks. Metadata also has optional fields to store other information such as file size and creation timestamp.
Client API has responsibility to create metadata message by analyzing the data which it wants to store. Metadata should be stored as sequenceNo 0.
For example, If a file has 8 chunks of data. seq_size should be 8 + 1. (1 for metadata chunk).

```
message Metadata {
    required int32 seq_size = 19;
    optional int64 size = 20;
    optional int64 time = 21;
}
```

Query message contains action to be performed. It may also contain key, sequence number, data or metadata. Store query should contain metadata or data with it. We have provided functionality to store string data without the key. It is a special case in which our system calculates key for the client and returns the key.

```
message Query {

    // Action to be performed
    optional Action action = 5;

    // Key to store; optional
    optional string key = 6;

    optional int32 sequence_no = 8;

    // Data in bytes to store
    optional bytes data = 7;

    optional Metadata metadata = 22;

}
```

Response message is reply from the cluster. It contains action for which cluster is responding. Depending on option it may contain data, metadata or just info message. It also contains failure which cluster can return in case of error or failure.

Response and Query messages are wrapped in CommandMessage as a payload.

```
message Response {

    // Action for which response is being sent
    optional Action action = 10;

    // Was that action successful?
    optional bool success = 11;

    // The key for which action was performed. If store action does not
    // provide key, return newly generated key.
    optional string key = 12;

    optional int32 sequence_no = 14;

    // Message, if any, after successful operation
    optional string infomessage = 15;

    oneof payload {

        // Failure with reason
        Failure failure = 16;

        // Data in case of GET action
        bytes data = 17;

        Metadata metaData = 18;
    }
}
```

Data Flow

There are three logical points in the flow of the data in the system.

1. Client API
2. Command Server
3. Work Server

1. Client API

Client API provides functionality for user to store and retrieve data. Presently client API support GET, GETS, PUT and PUTS functionalities. User can provide key and value to functions from command line. Client API communicates with Command Server. Internally, it uses message client object which is responsible for creating and writing different command messages.

For testing purpose, we have created a multi client runner class which creates a thread pool which will fire request to the server. Presently we tested with running 100 threads in batch of 10 threads. We identified that protobuf recommends message size should not exceed 1MB. So, we decided to chunk files which needs to be stored on the cluster. So we split files into 1MB chunks.

We have SerializationUtil class which takes care of file splitting and rejoining the chunks into a file. We also have written test cases for this in SerializationUtilTest.

2. Command Server

Command server is external server to which clients can communicate. Whenever command server receives a new message, message is forwarded to chain of responsibility in command channel handler.

Chain of responsibility handles message types such as Ping, Message and CommandMessage. If message is of type command message with query, we transfer it to the work server for processing.

For communication between work server and command server we have created two queues in QueueManager class. On each side, there is a worker thread which will fetch tasks from queues and direct them to correct place.

We have implemented this functionality considering that work server and command server are two threads which can communicate using shared queues. Our design also provides an opportunity to add more ask handler threads which will pass data from command to work or vice versa. This is important to handle large volumes of requests.

Command server also maintains two maps to keep track of message key and channel. These maps are used to reply back messages to the client.

Following is code snippet from CmdStorageMsgHandler which recevies data from work server and flush it back to the client. It waits for data from work server. Whenever it receives data, it fetches channel for client and writes data into it.

```java
@Override
public void run() {
    logger.info("Started Command Storage Message Handler...");

    while (forever) {
        try {
            // Getting message from work server. This message should be
            // forwarded to the client.
            CommandMessage msg = queues.getFromWorkServer().take();

            // Update Address of the source from where I recieved Message
            // Request...
            if (key2Address.containsKey(msg.getResponse().getKey())) {
                SocketAddress addr = key2Address.get(msg.getResponse().getKey());
                if (addr2Channel.containsKey(addr)) {
                    addr2Channel.get(addr).writeAndFlush(msg);
                } else {
                    // I assume if there should always be an entry in
                    // add2Channel map, if there is an entry in key2Add
                    key2Address.remove(msg.getResponse().getKey());
                }
            } else {
                logger.info("No Client is waiting for the response....");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3. Command Message to WorkMessage

Work server exchanges data in WorkMessage format. We have created a MessageAdaptar class which transforms WorkMessage to CommandMessage and vice versa.

We have modified Task message to contain CommandMessage. We prefered this since we wanted to save header information as well as query information originated from client. Adapter will create a new workmessage with task and put command message as a field of task. Now, this message can be passed to any node.

4. Work Server

Work servers are responsible for carrying out the tasks. WorkMessage containing task is transferred to leader through chain of responsibility pattern. We have developed AbstractTask which is generic task. GetTask is responsible for fetching data from remote nodes as well as self. It also returns data to the client. We have maintained set of delivered sequence IDs to solve duplicate message issue. This ensures that client will receive unique copies of the chunk. This also improves system response since chunks are returned as soon as they are available with leader.

WorkMessage is forwarded to the required nodes. In case of GET query, message is forwarded to all the nodes in the cluster. For STORE query, message is forwarded to the node IDs selected by replication strategy.

Follower nodes receive these work messages and pass them to handleCmdQuery() method of the Follower class. These messages are added to the TaskList.

5. Task Worker

Task workers are threads which monitor task list. Task list is a blocking queue which stores Task to be performed by the node. Task worker uses DBHandler object to perform database operations. We have created a ExceutorService which will hold these threads. TaskWorkers are started from StartWorkCommunication class along with the work server. We have created 4 worker threads; however, this value can be added to the conf files so that according to the load on the machine number of worker threads can be changed.

```
int workerCount = 4;
ExecutorService executors = Executors.newFixedThreadPool(workerCount);
for(int i = 0; i < workerCount; i++) {
    TaskWorker taskWorker = new TaskWorker(state);
    executors.execute(taskWorker);
}
```

This data is returned back to the leader which then identifies node from which it received the request. It then forwards this data to that node, which in the end forwards it to the client.

Data Replication

Replication of data in analogy with log replication in the RAFT algorithm means that the leader should accept the requests from client and replicate them across few nodes of the cluster to increase the availability of data. Data replication helps in preventing data loss in case of failure of one of the nodes from a cluster.

Advantages

1. Increased availability and reliability of data.
2. Fast retrieval of data in case of a large network.
3. Less movement of messages and data over the network when more number of replicas are present.

Disadvantages

1. Requires complex code to maintain consistent database.
2. The space consumption increases as same data is replicated on multiple nodes.
3. Any update of data requires carrying out the request on the available replicas which increases the overhead of performing the updates.

Implementation in our system:

The implementation of data replication in our system is achieved by having a Round Robin strategy which implements the common interface *IReplicationStrategy*, having a method to get the node ids where data should be replicated.

```
public interface IReplicationStrategy {

    List<Integer> getNodeIds(List<Integer> activeNodes);

}
```

The Round Robin strategy is implemented as below. The *size* variable contains the number of nodes, on which the data should be replicated. In our implementation, we have fixed it to 2.

```
@Override
public List<Integer> getNodeIds(List<Integer> activeNodes) {
    List<Integer> output = new ArrayList<Integer>();
    int temp = 0;
    index++;

    if(activeNodes.size () < size)  {
        return activeNodes;
    }

    while (output.size() < size) {
        output.add(activeNodes.get((index + temp) % activeNodes.size()));
        temp++;
    }
    return output;
}
```

After getting the node ids, the leader then follows the steps mentioned in Data Storage section.

To test the Round robin strategy we have written JUnit test cases (*RoundRobinStrategyTest.java*) and executed them successfully.

Work-Stealing:

To maximize the system performance, we should opt for work stealing. We have implemented a method *shouldSteal* as below which will check if the node should steal the task. Task Worker thread, before going in wait for Task in TaskList, checks if it should steal task from other nodes. *SIZE_FOR_STEALING* is the allowed number of enqueued tasks in an inbound queue before work stealing should be done.

```
public boolean shouldSteal() {
    return inbound.size() < SIZE_FOR_STEALING;
}
```

If TaskWorker needs to steal task, it will call startStealing method. We have not implemented this method yet. However, this method will send a STEAL_TASK message to all the edges.

```
@Override
public void run() {
    logger.info("Starting task worker : " + Thread.currentThread());
    while (forever) {
        if (state.getTasks().shouldSteal()) {
            startStealing();
        }
        Task task = state.getTasks().dequeue();
        CommandMessage msg = task.getTaskMessage();
        Query query = msg.getQuery();
```

We have implemented BasicRebalancer class which is used for deciding whether node should allow rebalancing. Node is allowed to delegate tasks if TaskList is filled more than 50%.

Closing Remarks

- ✓ Developed and implemented a new approach to store and find data in distributed system.
- ✓ This system is highly dynamic and enables to add additional servers dynamically.
- ✓ No dependency between the architecture of each server, servers are loosely coupled.
- ✓ Servers are not built on a assumption of single Software Stack. Each server can work with different data store.

Future Scope:

- ✓ This system can be extended to maintain the versions of the data being stored.
- ✓ This system is currently built on message broadcasting, it can be further enhanced to improve the efficiency of network.
- ✓ This tool can be deployed on the cloud and Rest API's can be developed to connect to server.
- ✓ Encryption algorithms can be implemented to improve the security of data being transferred over network.

References

- ✓ D. Thakur, "What is Data Replication? Advantages & Disadvantages of Data Replication.," E-Computer Notes, 2014. [Online]. Available: http://ecomputernotes.com/database-system/adv-database/data-replication. [Accessed 6 November 2014].
- ✓ N. Maurer, "Why Netty?," 2014.
- ✓ Diego Ongaro and John Ousterhout, " In Search Of an Understandable Consensus Algorithm", Stanford University
- ✓ CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In Proc. OSDI'06, USENIX Symposium on Operating Systems Design and Implementation (2006), USENIX, pp. 205–218
- ✓ BOLOSKY, W. J., BRADSHAW, D., HAAGENS, R. B., KUSTERS, N. P., AND LI, P. Paxos replicated state machines as the basis of a high-performance data store. In Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation (2011), USENIX, pp. 141–154.
- ✓ https://github.com/allengeorge/libraft/blob/master/libraft-core/src/main/java/io/libraft/algorithm/RaftAlgorithm.java
- ✓ http://thesecretlivesofdata.com/raft/
- ✓ https://www.elastic.co/blog/found-leader-election-in-general
- ✓ http://netty.io/
- ✓ https://github.com/google/protobuf
- ✓ https://developers.google.com/protocol-buffers/
- ✓ https://github.com/xetorthio/jedis/wiki/Getting-started