**SeattleU File System**

**Team**

**[Troy Osland**

**Jingyi Zhang**

**Domnic Savio**

**Sowmya Satyavada]**

**TABLE OF CONTENTS**

1. Introduction 1

1.1 Problem Statement 1

1.2 NEEDS STATEMENT 1

2. Proposed TECHNICAL APPROACH 2

2.1 COMPONENTS 2

2.1.1 NAMENODE 1

3. CONFIGURATION PARAMETERS 2

4. CLIENT IMPLEMENTATION DESIGN 2

5.DataNode & Namenode Implementation 2

6. Running Seattleu file system AND EXPECTED OUTPUTS 2

# Introduction

This is the description for SeattleU File System. This is a replication of HDFS (Hadoop File System) architecture.

## Problem Statement

Create a File System that mimics the Hadoop File System for Seattle University students.

## Needs STATEMENT

Have a file system that can be used to store and retrieve files.

# Proposed TECHNICAL APPROACH

**2.1 COMPONENTS:**

The proposed components in our solution. These components are going to be very similar to the HDFS components and will provide similar functionality.

**2.1.1 NAMENODE:**

The Name Node, also called as the Master, stores all the namespace, edit log information of all files in the file system, and directory structure. It provides the path to the data nodes where the actual files reside.

* The Name Node runs on a single EC2 instance in AWS.
* Supported commands include opening closing and renaming files in the namespace.
* When Name Node is started, it creates a directory structure. It supports directory management commands, and store directory information to a backup after every change.
* Name Node never initiates communication and always responds to Client and Data Node requests.
* Inspect the size of Data Node before sending them any data.
* Every data recorded by the Name Node is forwarded to a Name Node backup.
* The Name Node backup periodically checks if the Name Node is alive, and if it finds the Name Node doesn’t respond to its requests, it declares itself as the new Name Node.
* When the Name Node stops receiving heartbeats for a Data Node it assumes that the Data Node is no longer available and the blocks on that Node are not available. It responds to other Data Nodes to create new replicas of the blocks.

**2.1.2 DATANODE:**

The Data Node stores the actual data in their systems. The features include.

* On Startup Data Nodes register with the Name Node and get a unique Id which is also their registration Id.
* Periodically Data Nodes send heartbeat information (add separate section for heartbeat) to Name Node with information including space left on device, activity on the device.
* Periodically Data Nodes also send block reports, information about the number of blocks, number of block replicas that are stored on the node.
* Every time a Data Node receives a block, it notifies the Name Node of a block received.

**2.1.3 CLIENT:**

The client provides the file system abstraction to the end user, providing the functionality of putting and getting files and hiding the notion of Name Node and Data Node to the end user.

**2.1.4 APPROACH:**

The approach will be to use python to describe the name and data nodes. The name and data nodes will be running on ec2 and support communication over RPC which communicates over SSH using TCP, similar to HDFS. We will also have a python client module that will run locally and provide an interface to the SUFS running on ec2.

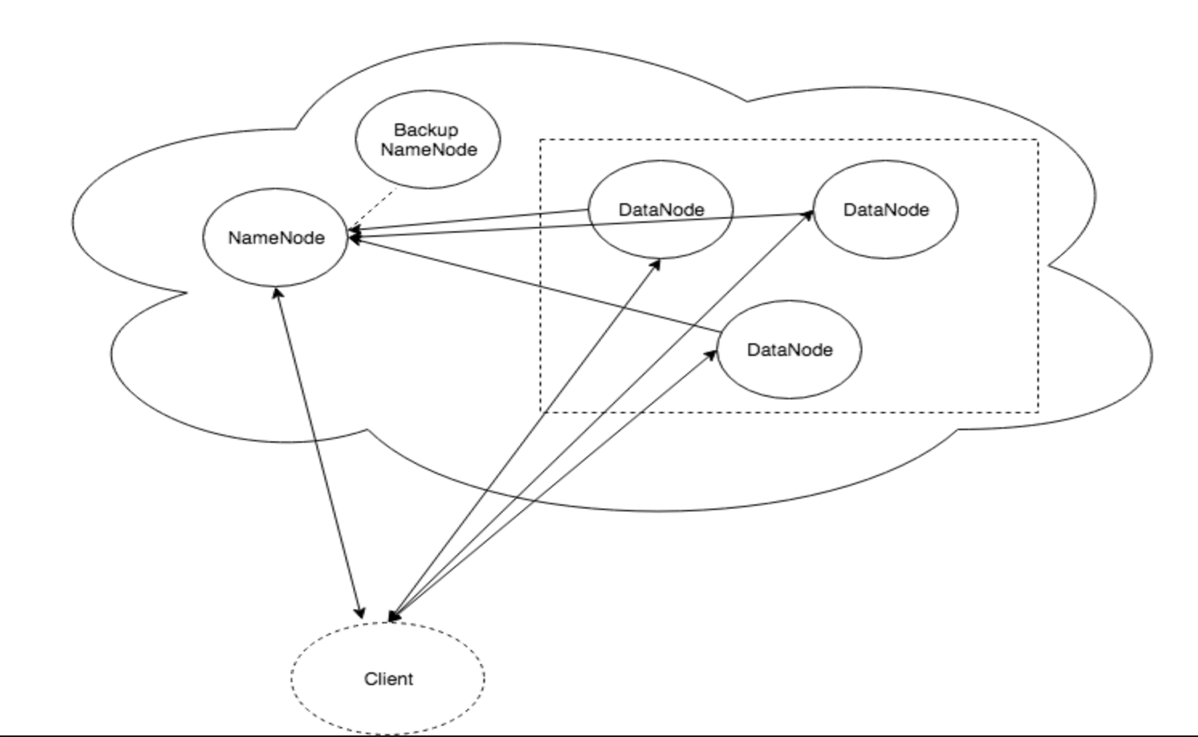
**2.1.5 ASSUMPTIONS:**

* Replication Factor is same for all files and configured at startup.
* If the Name Node goes down, the Data Nodes should be updated with the new IP of the Name Node which is stored in configuration files on the Data Node machine.
* In order to guarantee replication and consistency, number of Data Nodes should be more than the replication factor at all times.

## Requirements

* A Distributed File System running on EC2
* Architecture similar to HDFS
* Support Name Node and Data Node models.
* Ability to put files on the File System.
* Ability to get files from the File System.
* Ability to delete files from the File System.
* Provide support for putting and getting directories.
* Secondary Name Node support in-case the Name Node fails
* Data Node Resiliency.
* Read input data from AWS S3

## Architecture Design For SUFS



**2.3.1 Technical Differences b/w HDFS and SUFS:**

* SUFS Name Node does not support storing namespace changes in edit logs periodically.
* The replication factor can be configured at the system startup time and cannot be configured on a per file basis.
* If a Data Node goes down, a manual intervention is required to start another Data Node. Once the Data Node is started it automatically joins the Data Node cluster.

**2.4 COMMUNICATION PROTOCOL**:

* We used Python’s RPYC (<https://rpyc.readthedocs.io/en/latest/)> to establish communication between all components i.e. Client Name Node and Data Node.

**3 CONFIGURATION PARAMETERS:**

* **3.1.1 CONFIGURATION FOR NAMENODE:**
  + Replication Factor: 2
  + Block Size: 10
* **3.1.2 CONFIGURATION FOR DATANODE:**
  + Name Node Address: <host>:<port>
  + isPrimary = 1
* 3.1.3 **CONFIGURATION FOR SECONDARY NAMENODE**
  + **Secondary** Name Node Address: <host>:<port>
  + isPrimary = 0

**4 CLIENT IMPLEMENTATION DESIGN**

**4.1.1 Client – Name Node:**

* Create files in namespace.
* Initiate file read requests.
* Initiate write requests.

**4.1.2 Client – Data Node:**

* Buffer data until the block size and send write the block to the data node.

**4.1.3 READ from file system**

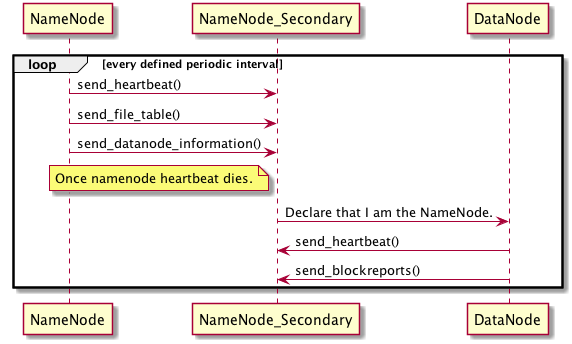


**5.1.4 WRITE to file system**



### 5.1.5 DELETE from the File System /Directory Structure

### 

****

### 5. DATANODE & NAMENODE IMPLEMENTATION

### 5.1.2 REGISTRATION WITH NAMENODE

### 

### 6.1.2 PERIODIC HEARTBEATS

### Heart Beats are currently sent by the Data Node every 30 seconds. Identify disconnected Data Nodes from cluster.

### Name Node checks every 45 seconds, for Data Node to see if all the heartbeats

### 

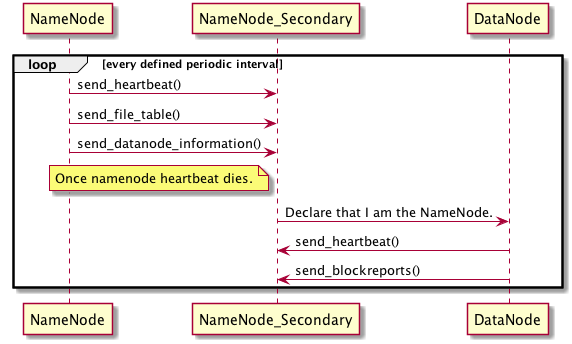
### 5.1.3 PERIODIC BLOCK REPORTS

### Data Node sends block reports once every 2 minutes. Periodic Block Report check and rebalance blocks with less than replication factor. Name Node checks every 2:30 seconds, for the existing.



**5.1.4** **SECONDARY NAMENODE IMPLEMENTATION**

Name Node sends heartbeat to Secondary NameNode periodically and also send block data, file table info and directory structure.When the NameNode dies .Secondary NameNode announces its ip to all DataNodes to connect.



**6 API & DATASTRUCTURES**

**Client to Name Node:**

# *send adding file to directory request to Name Node with path provided by user*

**def add\_dir(master,path)**

# *send listing content in directory request to Name Node with path provided by user, and print the contents received from Name Node*

**def list\_dir(master,path)**

# *send deleting directory request to Name Node, and delete all files under that directory*

**def delete\_dir(master,path)**

# Specify the filename that the client intends to read from SUFS

**def read(filename)**

# Specify the file to store in SUFS

**def put(localfile,filename)**

**API Exposed on the Name Node:**

# This method takes a file name and returns to the client the already stored mapping for all the blocks of the file and corresponding datanodes the blocks are in.

**def exposed\_read(self, fname):**

# Periodically called by the datanode with its registered data node id to show health status.

**def exposed\_receive\_heartbeat(self, datanode\_index, msg):**

# Peridically called by the data node with all the block information of the corresponding data node.

**def exposed\_receive\_blockreport(self, datanode\_id, blockinfo):**

# Called by the data node on startup to register itself and its address.

**def exposed\_register\_datanode(self, ip, port)**

# Allocate the blocks based on the size of the file and return the block mapping for the destination file name.

**def exposed\_write(self, dest, size)**

# returns the file table entry which has the mapping for all the blocks of the corresponding file name.

**def exposed\_get\_file\_table\_entry(self, fname)**

# *add directory and store the change to backup*

**def exposed\_add\_directory(self,path)**

# *return all file names under given directory*

**def exposed\_get\_delete\_filenames(self,path)**

# *delete all files under given directory*

**def exposed\_delete\_directory(self,path)**

**API Exposed on the Data Node**

#This method is a response from the namenode when it wants the block to be replicated on a new data node once an exisiting data node dies.

**def exposed\_replication\_forward(self, block\_uuid, datanode)**

#Store the block data for the corresponding block id and forward the block to the datanodes

**def exposed\_put(self, block\_uuid, data, datanodes)**

# returns the block data for the block id.

**def exposed\_get(self, block\_uuid)**

**API Connecting to S3 and reading file**

**# def put\_s3(master,bucket,key,dest):**

s3 = boto3.client("s3") , obj = s3.get\_object(Bucket=bucket, Key=key)

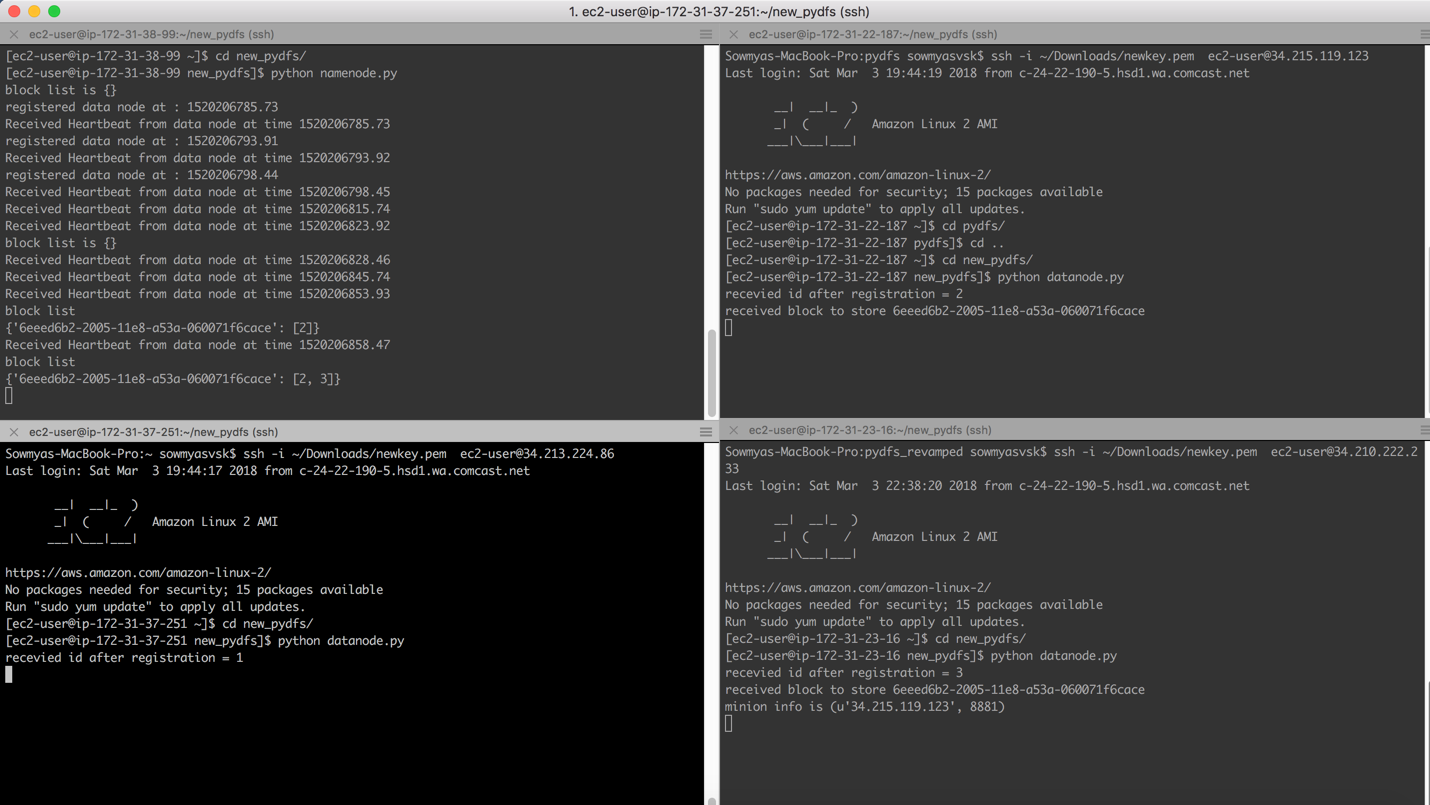
size = obj['ContentLength']

# 7 Running Seattleu file system AND EXPECTED OUTPUTS

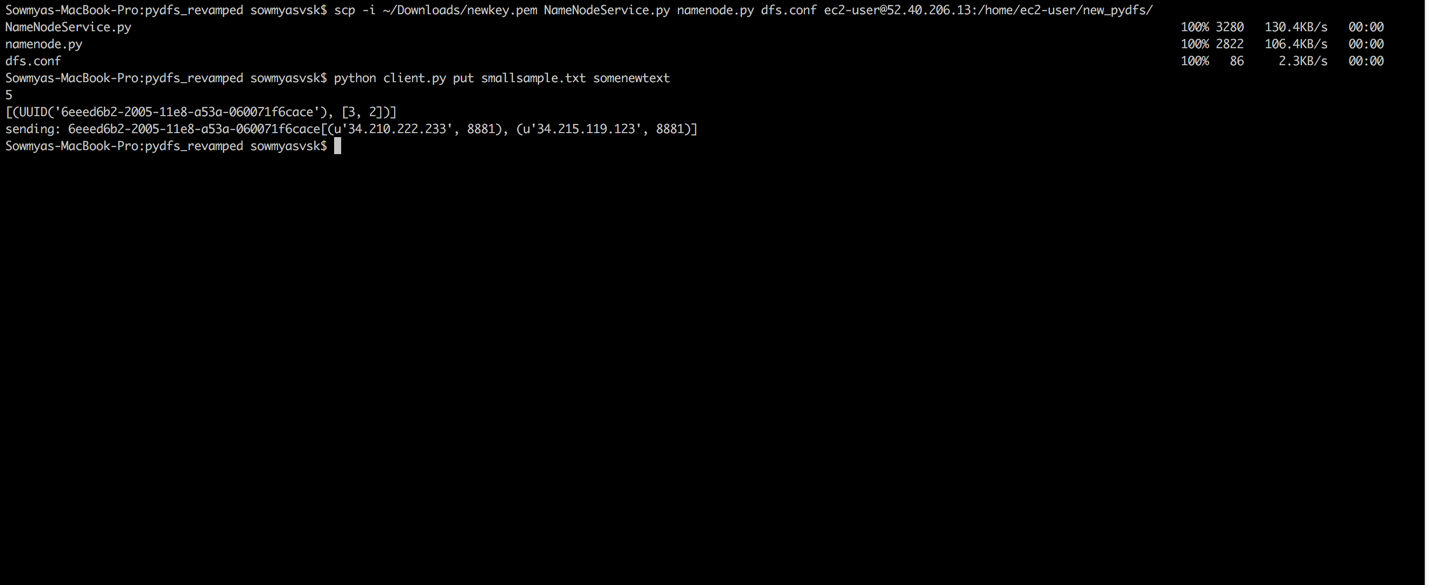
See the README for instructions to run SUFS.

* SUFS running on EC2.
* Put files using the clients input as S3 file
* Get files using the clients.

EC2 instances with Name Nodes and 3 Data Nodes



Client on local machine



Removing one of the existing Data Node

## sufs_project_files/Screen%20Shot%202018-03-04%20at%203.42.12%20PM.png

Secondary Name Node