HDFS Overview

The Hadoop File System is an important subcomponent to Hadoop. Hadoop was built to rival Google's BigTable and contains various subcomponents that come together to form Hadoop's ecosystem. These subcomponents include HBasem a NoSQL Database, MapReduce, a distributed processing application, Hive, a query application, Mahout, a machine learning application, Oozie, a workflow and scheduling application, ZooKeeper, a coordination application, and Ambari for management and monitoring of the cluster. Hadoop File System is an important piece to this puzzle as it allows for the storage of data in a cheap yet reliable system. Many companies rely on this software for these reasons, and in the following, the internals and configuration of the HDFS will be explained as well as a performance report under a variety of different conditions.

HDFS is necessary for companies with a lot of data that need to store data reliably yet cost effectively. Hadoop File System is completely open source, meaning anyone can add to it and use it, and more importantly, it runs on commodity hardware. Commodity hardware are cheap computers strung together to make servers that often fail and are unreliable. Each one of these cheap computers is referred to as a node or DataNode, and the main computer that manages all the nodes in the system is referred to as the master or NameNode. On a macro level, HDFS consists of many DataNodes which are managed by the NameNode. The underlying requirement for a machine in the cluster is that it supports Java, but other than this there are limited requirements. This means that unlike a homogeneous distributed data system, HDFS is able to be made up of a medley of different computers.

Files in the HDFS are held in blocks. HDFS determines a block size, and then splits files up into appropriate sizes. Blocks are distributed across the system a minimum of three times to prevent hardware failure from hindering the cluster, however by increasing the replication factor in the cluster this can be increased. This block redundancy is the key to HDFS's ability to work on unreliable hardware.

As the name suggests, the data node holds the data in the cluster where as the NameNode holds all the metadata needed to direct the cluster. The NameNode has various responsibilities in managing the Hadoop File System. One such responsibility is to receive Heartbeat data and block reports from DataNodes so the NameNode knows which DataNodes are functioning, another responsibility is to check if all the data in the cluster is appropriately copied in the cluster. Furthermore, one last responsibility of the HDFS is to record all the pertinent metadata in the cluster, these include the location of stored block, file size, the folder structures, etc. Two important files on the NameNode are the FsImage, which tracks the file system namespace, and the EditLogs which contains file system changes.

The DataNode which runs on commodity hardware has fewer responsibilities than the NameNode. The DataNode's responsibilities include maintaining daemons, storing blocks, perform small read or write requests made by the client, and sending periodic heartbeats so that the NameNode knows they are still functioning.

The Secondary NameNode. The secondary NameNode writes data from the NameNode's RAM to its harddrive and is responsible for combining information stored in the EditLogs and FsImage together from the NameNode. This is done by periodically applying information from the EditLogs to the FsImage and then copying the newly created FsImage back to the NameNode.

Performance Analysis of HDFS:

Although one might intuitively think that a file system would work better with small files, HDFS is optimized for large file sizes. Partitioning large files into smaller ones can hinder the system as it adds unnecessary extra files in Hadoop's NameNode file structure. General rules of thumb exist for file size to optimize a user's usage of HDFS. It is recommended that each file is at least a minimum of 1GB. This is primarily to ease the work for the NameNode. With many small file sizes, the NameNode must record more information which at scale can hinder the cluster.

HDFS uses “write once”/“read many” workflow. What this means is that, while we cannot edit files that are already stored on the HDFS system, we are able to append to the files by reopening and adding to the data. In this kind of appending operation, the client begins by interacting with the NameNode.The NameNode retrieves all of the file blocks from the DataNodes and reassembles them. Only then is the client able to append to the file. The file is then redistributed with the new data appended into new blocks. While initial reading and writing have similar performance, it is easy to see how an appending process is computationally expensive.

More DataNodes will often increase performance across an HDFS system. However, the speed of read/write processes is not the only thing that needs to be considered when designing a distributed file system. A larger system will be more expensive, so it would be best to keep it as small and efficient as possible given the size of your data set. A small data set on a very large system with large amounts of replication and very small block sizes is not worth the investment as it would be a bloated, computationally expensive system.

HDFS Clone Project Overview

The HDFS Clone project was built and tested on a Windows stack computer network. The machines had already been preconfigured to be discoverable to each other on the system by a local network. They were also all a part of the same internet network and only visible when logged on to the VPN. Despite the constraints, the HDFS Clone should work on other systems.

The architecture and use of our hdfs\_clone project is simple by design. It contains three main components. Master, Nodes, and Client python files. Every machine operating in the system will share the same source code for the project, but they will use the code differently depending on the intended function of the machine.

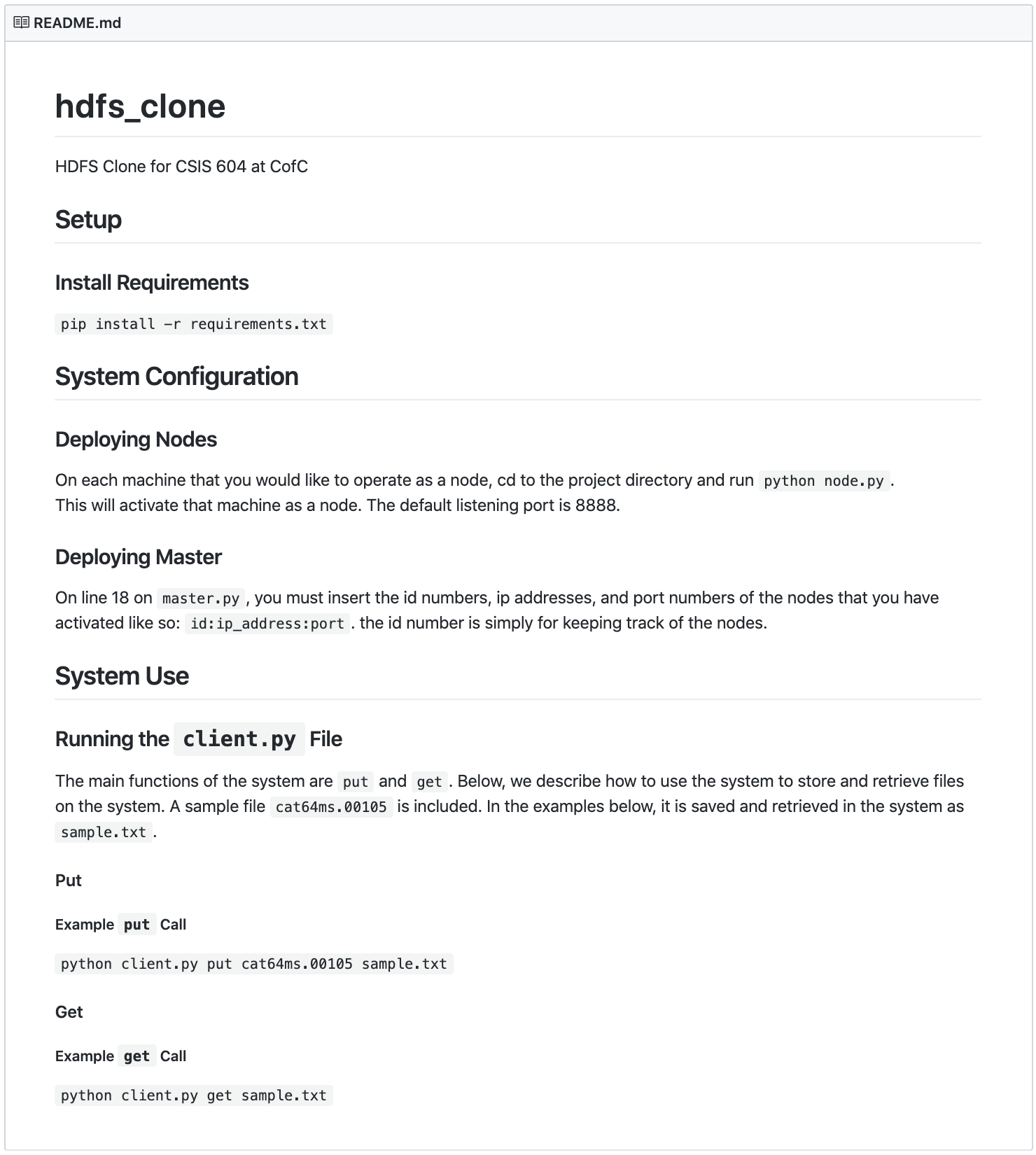
The machine that is going to be used as the Master node needs to run the master.py script. This script will activate the machine as a master node on a specified port. It waits for commands from a client, executing them across the nodes. On the master node is the logic that divides up the file to be sent to the data nodes or reassembles the file after it has been retrieved from the data nodes. It assigns and maintains records of where the file blocks will live across the system.

Machines that are to be used as nodes should run the node.py file. This script opens a port on the machine to receive commands and data packets from the master node. The data nodes contain the logic to store and retrieve the file blocks locally and send them back to the master node.

The client.py file acts as our client through a terminal interface. It is called by invoking the python command followed by the file name and various parameters. The script interacts with the exposed master node in executing its commands.

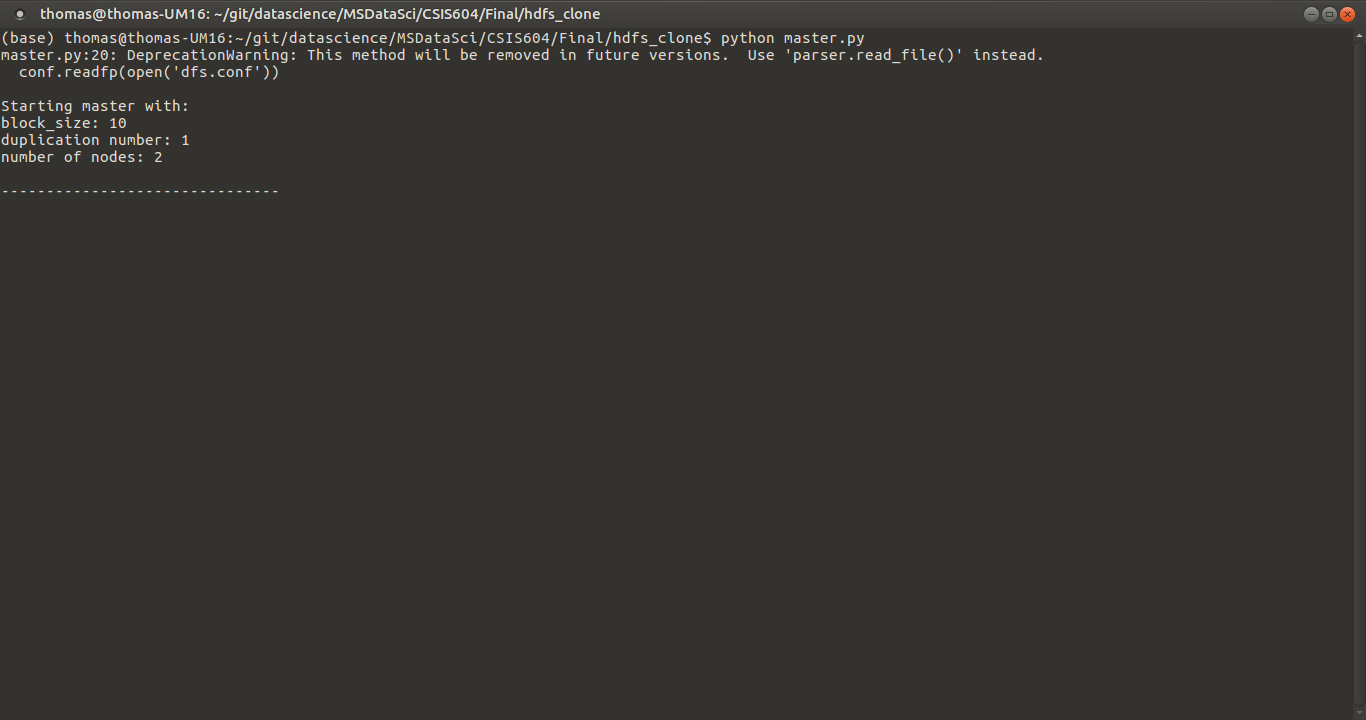
The project uses a library called RPyC, or Remote Python Call. It acts as the main engine for our HDFS clone and is typically used for distributed computing and remote procedure calls. It is incredibly useful for this project because requires no special decoration or definition languages. This makes it very lightweight when using in a simple proof of concept like we have created. Additionally, the project includes a requirements.txt file to aid in setup, a \tmp directory, and a README.md file

Supporting Documentation and User Guide

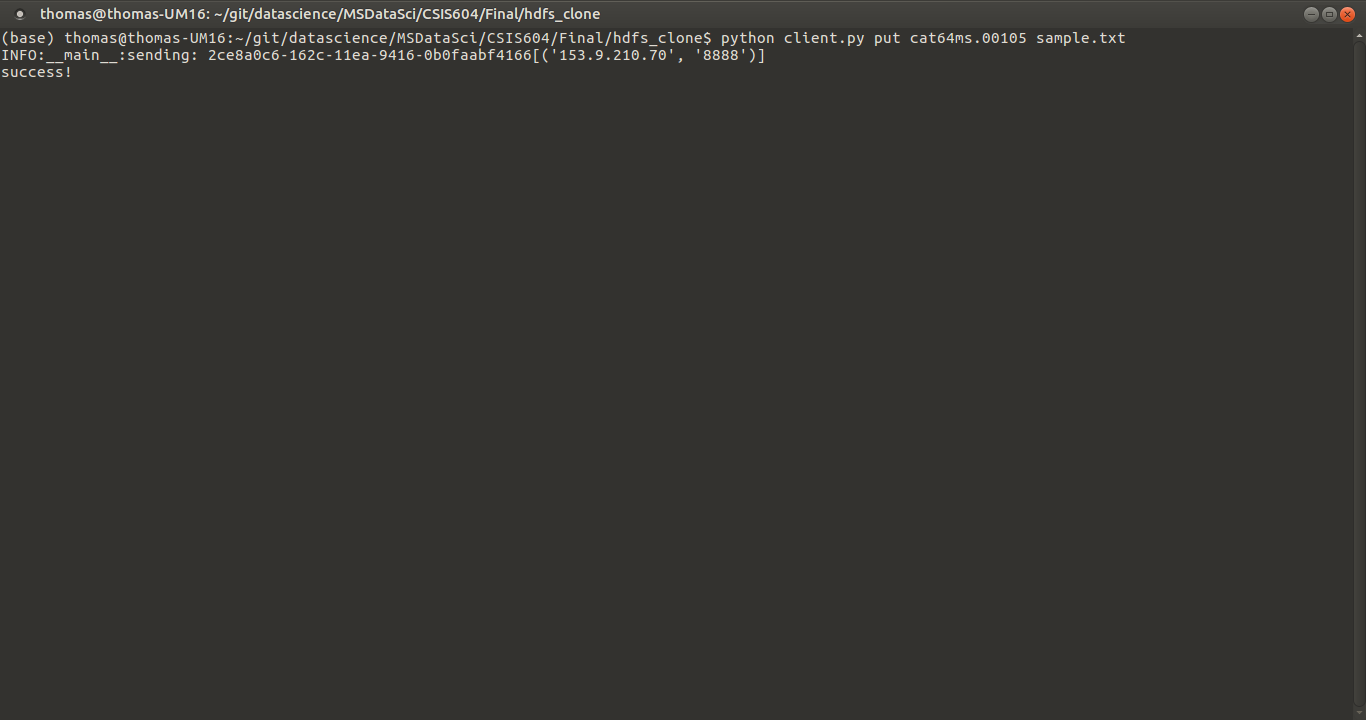


Sample Shell Command Usage

Master:



Client:



Node:

