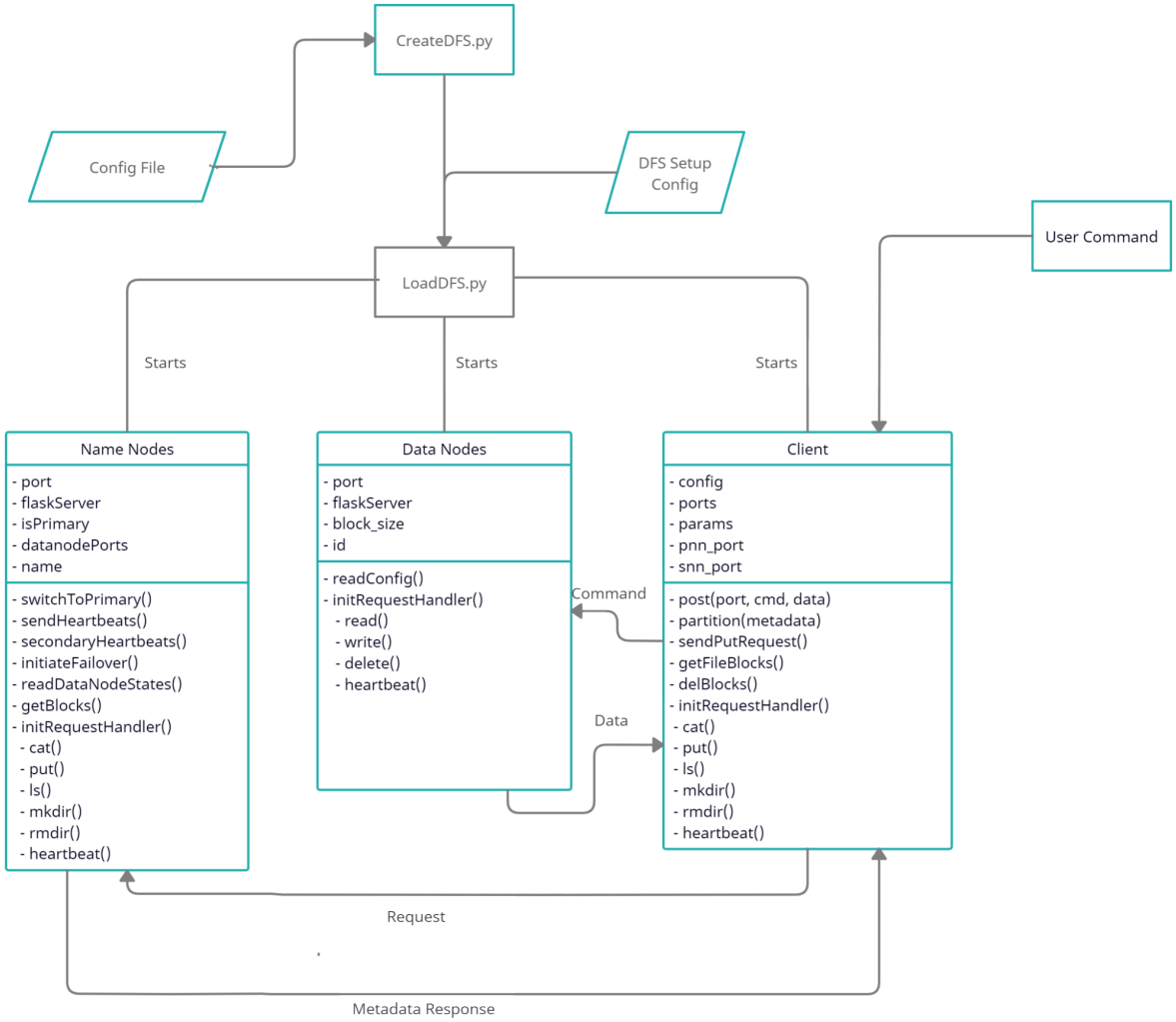
Project Title - Yet Another Hadoop

Problem Statement - The goal of the project is to simulate a mini hdfs setup on a local system by simulating the working of namenodes, datanodes and the replication of data blocks across multiple datanodes.

Design Details -



Implementation Details

Key Features:

1. Multiple DFS simultaneously
2. Persistent Storage
3. Actual Running processes and servers
4. Fault tolerant to deletion of datanodes and namenodes
5. Namenode checkpointing
6. Robust client (in case of namenode down, waits until backup namenode is up)
7. Map reduce on DFS files

Details:

- *Create_dfs.py* is run first. This reads the config file, makes a dfs_setup_config file, and checks if all the paths and directories to the namenodes and datanodes exist.
- *load_dfs.py* starts a new subprocess for the namenodes, datanodes and also starts the client. It needs to be noted that if no config file path is given in the arguments, the previous config file is used. Each namenode and datanode has its own flask webserver running on a unique port.
- Once the *client* is set and running we can now use it to execute commands like put,cat,rm,mkdir,rmdir,ls.
- The *client* reads the commands from the user and makes the necessary requests to the namenode server to get relevant metadata. Using the metadata it can then form requests to the datanodes to read/write/delete the data
- The requests are sent as HTTP GET/POST requests

.

- *The working of namenode.p*y - Namenode is capable of starting a flask server to listen to the requests from the client. It also runs a thread continuously to monitor heartbeats to the datanodes and waits for a response, and keeps updating the status of the datanodes.
- The Primary Namenode writes the operations and the data to an edit log. The Secondary Namenode reads the edit logs periodically and applies the edits to its own fsimage
- The fsimage is represented using a directory tree with files on the underlying os. These files do not contain data and instead only contain metadata. The actual data is stored on the datanodes
- The secondary namenode is capable of sending heartbeats to the primary. If it doesn't get a response, it takes over as the primary namenode and starts another secondary namenode process by updating the port numbers,path to primary and secondary namenodes.

- *Working of the Datanode* - Datanode performs 3 actions - read,write,delete. In the read requests, it just reads the file using the name and returns it. For write and delete, it uses gossip where the datanode inserts/deletes the file in it and removes itself from the list of datanodes, now sends the list to the next datanode recursively.

- *Mapreduce* uses a cat request to get the file contents from the datanodes, then uses another subprocess to run the map.py, sort it and pass it to reduce.py using a shell command. The results,status are then written to hdfs using put request.

Reason for design -

We have decided to use the local file system itself to store the hdfs files and directories. This simplified code to check for path validity. We decided to use the different subprocesses for the namenode and datanodes as the communication would be easier, we can just send messages to ports. This was also the reason for choosing flask servers as the nodes. Mapreduce writes the output of the reducer to a temporary file which is then used to put into the hdfs. Status of the map reduce job was also inserted the same way as the put function takes a file as input.

Takeaway from project - We learnt about the internal working of hdfs and the specifics like heartbeats, gossip etc. We learnt to think about the architecture and use software engineering principles to complete the project. We also learnt about running subprocesses and communication between them. Lastly, we learnt how to coordinate as a team and debug code better.