Distributed Systems (CS 417)
Project-2 – Distributed File System

Preface: Projects 2 and 3 will be two phases of one big project. While writing code for this project, note that this will be used in the next project. Also, the choice of programming language in this part will get locked for the next part as well.

We have assigned you to groups of 2. We have a **“no divorce”** policy for the rest of this course. That is, you cannot change your group afterwards.

Recitation: I will be talking extensively about HDFS and the assignment in my recitation. Attending the recitation will help you in understand it better.

Project Statement

The first phase of the project requires you to implement a distributed file system similar to HDFS (Hadoop Distributed File System).

Motivation

In many environments, a lot of data needs to be readily available (fast access) and resilient to disk and system failures. Keeping in mind the large amount of data that needs to be stored (petabytes and exabytes):

1. No one disk can store all of it.
2. Keeping a complete file on one disk may bottleneck access speeds.
3. The expected number of disk failures per day in a large datacenter is not negligible, even with SSDs.

To cater to all these needs, HDFS was created. Its design is based on the Google File System (GFS), which was initially created to store data for Google search but found use in non-production environments as well. HDFS is used for big data storage throughout the world, notably by Facebook and Twitter. It:

1. Divides all its files into small chunks.
2. Stores $ReplicationFactor number of copies of each file chunk on different DataNodes.
3. Keeps all metadata (information about the files) in a fail-safe and highly available (low downtime) master node called the NameNode.

References

1. HDFS: https://storageconference.us/2010/Papers/MSST/Shvachko.pdf
2. HDFS Architecture Guide: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
3. The Google File System: https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

Details

The file system would be built using two major components:

1. NameNode (NN):

   This is responsible for storing information about the files in the system and handling the communication with DataNodes.

   When queried, the NN should be able to perform all major file operations such as *open*, *close*, *list* and should support a block management mechanism similar to HDFS.

2. DataNode (DN):

   Should be used for performing *read* and *write* operations of blocks.

3. Client:
   a. This is a command line interface to interact with the NameNode.
   b. Usage:
      i. `$> get hdfs_file local_file` ##Writes a file to the local filesystem from HDFS
      ii. `$> put local_file hdfs_file` ##Writes a new file to HDFS from local file system
      iii. `$> list` ##Displays all files present in HDFS
   c. Handles reads and writes from DNs on receiving metadata info from NN about a file.

Key Points

- Use RMI (java), RPC (C/C++), or RPyC (Python3.6) for communication.
- Server should be multithreaded for seamless communication
- HDFS should persist its state even in the event of NN restart
- DNs should also support restart and all operations should be carried on ordinarily after NN server is up and running again.
- NN should print a warning message if some DN(s) are down (no heartbeats) and some file is not readable due to it.
- Block replication Factor: Two way.
- Block size: 64Bytes (should be configurable in your configuration file) [Mention its name it in your README]
- You should use a configuration file to help change different run-time parameters without recompiling the source.
- All the parameters and return objects should be marshalled and unmarshalled from byte arrays (Using Google Protocol Buffers)

Reference Implementation Guide

All the APIs take a `byte[]` as an input parameter and return a `byte[]` as an output. The input and output `byte[]` arrays should be built over Google Protocol Buffers.

- ***`open`***
  - Create and return a unique handle for each opened file.
- ***`write`***
  - Write contents to the assigned block number
  - Workflow:

    `openFile(“filename”, ‘r’)`: (throw error if the filename already exists)

    in a loop:
    1. Call *`assignBlock()`* using handle from *`openFile`*
    2. Obtain a reference to the remote DataNode object using the first entry in the DataNode location

        3. Call `writeBlock()` on all the assigned DataNodes

        `closeFile()`

- <u>`read`</u>
    - Get all block locations for the file
    - Read blocks in sequence
    - Workflow:

      `openFile("filename")`

      in a loop:

        `getBlockLocations()` using handle from `openFile`

        Obtain a reference to the remote DN object using an entry in the DataNode location.

        Issue `readBlock()` to the DN (if this fails, try the next DN)

        Write to the local file.

      `closeFile()`
- <u>`Delete`</u>
    - Check if the file exists in HDFS: (Throw NotFoundError)
    - Delete the NN entry for this file. (This makes the file in accessible to user)
    - Send messages to each DN to delete corresponding blocks.

a. To keep things simple, we do not require you to do update-file or directory operations.
b. All remote responses should contain a status message depicting success or failure of the request.
c. Heartbeats from a DN should be periodically received by the NN to keep a check on the status of its health.
d. NN should be able to receive all the information about all the blocks in each DN using the function *blockReports*()
e. For each file, the NN just persists the filename, list of blocks associated to that file and its creating time. It doesn't store the locations of those blocks. Once DNs send *blockReports*, NN then knows the locations of each block and tracks this information in memory.
f. Clients and DN discover the NN from a conf file and read from a standardized location (/tmp/somefile). The conf file contains the socket information of the NN (port number).
g. Design the message protocol using protobuf. That will help you in standardizing your work across functions and files.
h. Write extensive comments; these will help you in filling in the code later and for others (including the graders) to understand your logic.

Possible Cluster Setup Configurations

- Each of the two group members can have one virtual machine installed by using VMware or other virtualization tools (docker, VirtualBox).
- You'll then have four machines for the HDFS cluster (2 VMs and 2 host machines).
- Connecting the machines with either a local Wi-Fi connection or LAN would enable all these machines access each other.
- Make sure you establish a working configuration way early before the deadline and test that the systems can communicate with each other.
- You can also use the following ilab machines instead of VMs: {kill, cp, ls, less}.cs.rutgers.edu . https://www.cs.rutgers.edu/~watrous/dcs-ilab-profile.html

Submission Rules

a. Put all your files in a folder `Project2_netid1_netid2` and create a zip file of that folder: `Project2_netid1_netid2.zip`
b. Write a txt/md file which illustrates your learnings from this assignment. Include the division of work between each team member.

c. Write a `README` which gives clear directions of how to compile & run your code. Examples will be helpful. Describe any dependencies you have in your source.

d. Create a `Makefile` that compiles your code.

e. The file tree for your submission should be in the following format: (Skeleton provided)

```
MAPR
 ├── Lessons.md
 ├── Makefile
 ├── README
 ├── bin
 └── src
     ├── com
     │    └── google
     │         └── protobuf //All Files
     ├── dn_config.txt
     ├── ds
     │    ├── hdfs
     │    │    ├── Client.java
     │    │    ├── DataNode.java
     │    │    ├── IDataNode.java
     │    │    ├── INameNode.java
     │    │    ├── NameNode.java
     │    │    ├── NetworkTest.java
     │    │    ├── README
     │    │    ├── hdfs.proto
     │    │    └── temporaryInterface.java
     ├── nn_config.txt
     └── permission.policy
```

f. Note: replace .java with .c / .py. Also, please add a she-bang (with specific python version) in all .py files.

g. Your bin file should be empty in the submission. It will be populated with all the binaries when compiled via the Makefile.

h. We shall use plagiarism checker for this project. Refrain from copying code samples from the web or your colleagues.

PS:

- Start this assignment as early as possible. The expected amount of code needed for this stage: ~2K.
- I shall be available in my office hours and recitations for any questions that may arise while implementation.
- The best policy is to get a bug free code is to write bug free code the first time. Do not start typing anything before you have decided the structure of the program.
- Clearly mark function names, their functions and their inputs/outputs in a pseudo code format. Make sure both teammates understand the structure so that they keep it in mind while implementing their portion of the project.
- Use of version control and web-based storage (github, gitlab etc.) for code becomes very important in large projects. Make a **private** repo for your team and regularly push your code there. (Fun Fact: Linux developers used to save versions in tarballs and patches before version control. Frustrated with such manual labour, Torvalds designed git: https://youtu.be/4XpnKHJAok8)